Philipps Universität Marburg

FACHBEREICH MATHEMATIK UND INFORMATIK
Arbeitsgruppe Grafik und Multimedia

Dissertation

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

Rekonstruktion und Verarbeitung von Objekten und Szenen aus Kamerabildern

Dipl.-Inform. Nico Grund
November 2012

Betreuer
Prof. Dr. Manfred Sommer
Prof. Dr. Michael Guthe

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße
35032 Marburg
Germany

Rekonstruktion und Verarbeitung von Objekten und Szenen aus Kamerabildern

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
am
Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg

vorgelegt von
Dipl.-Inform. Nico Grund
(geboren in Frankfurt/Oder)

Marburg, 30. November 2012

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein Straße 3, 35032 Marburg

Angefertigt mit Genehmigung des Fachbereichs Mathematik und Informatik
der Philipps-Universität Marburg

Gutachter:
Prof. Dr. Manfred Sommer, Philipps-Universität Marburg
Prof. Dr. Michael Guthe, Universität Bayreuth

Prüfungskommission:
Prof. Dr. Manfred Sommer, Philipps-Universität Marburg
Prof. Dr. Michael Guthe, Universität Bayreuth
Prof. Dr. Rita Loogen, Philipps-Universität Marburg
Prof. Dr. Ekaterina Kostina, Philipps-Universität Marburg

Abgabetermin am 30. November 2012.
Prüfungstermin am 22. Januar 2013.

Abstract

Highly detailed models are commonly used in computer games, animation films and other interactive rendering applications. In most cases the models correspond to a real object, which can be scanned using a laser scanner. Because modern laser scanners are expensive and unwieldy, alternatives are needed, which can generate a virtual duplicate in an economic and efficient way. In addition to the methods for the reconstruction of three-dimensional objects, intuitive editing techniques are also required. In this context multi resolution modeling algorithms are very popular to perform the editing process of a model in real time and thus simulate complex motion sequences, for instance.

This dissertation presents a novel solution for the reconstruction of three-dimensional objects and scenes using two-dimensional images and introduces new techniques for the real-time editing of a given polygon mesh. By using reciprocal pairs of images the correspondences between the individual pixels can be uncovered and thus a depth analysis can be realized. The resulting depth values are then stored in a depth map, which ultimately is used to generate a three-dimensional triangulated mesh. During the implementation of the procedure the parallelization of the individual steps of the calculations represents the most important part.

For the modeling of a 3d object a high-quality parallel simplification algorithm was proposed, which is able to create multiple resolutions of a complex triangle model in real-time. Based on this simplification procedure a parallel multi resolution modeling algorithm was finally realized, which can be used to modify the overall shape or small scale details of a 3d object at different levels. The completed modifications are propagated to all resolutions by preserving all details of the original surface. All required data structures were generated by a massively parallel simplification algorithm within a few seconds.

Kurzfassung

In Computerspielen, Animationsfilmen und anderen interaktiven Rendering-Applikationen werden für gewöhnlich 3D-Modelle verarbeitet. Die Modelle entsprechen dabei meistens einem realen Vorbild, welches mithilfe eines Laserscanners abgetastet wurde. Da moderne Laserscanner teuer und unhandlich sind, werden Alternativen benötigt, mit denen virtuelle Abbilder kostengünstig und effizient erzeugt werden können. Neben den Methoden zur Rekonstruktion dreidimensionaler Objekte sind Verfahren erforderlich, mit deren Hilfe die 3D-Modelle modifiziert werden können. In diesem Kontext spielt die Multiskalen-Modellierung (engl. *multi resolution modeling*) bei der Durchführung des Editiervorgangs in Echtzeit eine wichtige Rolle, um zum Beispiel komplexe Bewegungsabläufe simulieren zu können.

Diese Dissertation beschäftigt sich mit den Möglichkeiten zur Rekonstruktion von Objekten und Szenen aus Kamerabildern und präsentiert neue Techniken, mit denen ein als Polygonnetz vorliegendes 3D-Modell editiert werden kann. Für die 3D-Rekonstruktion werden reziproke Bildpaare verwendet, auf deren Grundlage die Korrespondenzen zwischen den einzelnen Bildpunkten aufgedeckt und eine Tiefenanalyse vollzogen wird. Die daraus resultierenden Tiefenwerte werden in einer Tiefenkarte (engl. *depth map*) gespeichert, aus denen letztlich ein dreidimensionales Dreiecksnetz generiert werden kann. Während der Umsetzung des Verfahrens wurde großer Wert auf die Parallelisierung der einzelnen Berechnungsschritte gelegt.

In Bezug auf die Modellierung von 3D-Modellen wurde zunächst ein hoch-qualitativer, paralleler Simplifizierungsalgorithmus entworfen, der in der Lage ist, in Echtzeit mehrere zu einem 3D-Objekt gehörende Detailstufen zu erzeugen. Auf Basis des Simplifizierungsverfahrens wurde schließlich ein parallel auf der Grafikkarte ausführbares Programm zur Multiskalen-Modellierung realisiert, mit welchem die Möglichkeit geschaffen wurde ein Modell auf verschiedenen Detailstufen zu editieren und die vorgenommenen Modifikationen über die erstellten Detailstufen hinweg in Echtzeit und unter Berücksichtigung der bestehenden Oberflächendetails zu verarbeiten. Die für das Editieren notwendige Datenstruktur wird dabei während der Simplifizierungsphase parallel auf der Grafikkarte innerhalb weniger Sekunden erzeugt.

Vorwort

Das Leben stellt einen immer wieder vor die Qual der Wahl. Menschen, die aus den Bauch heraus einen bestimmten Weg einschlagen, werden unter Umständen als Lebenskünstler oder Glückspilze bekannt. Andere, die sich kämpferisch keinem Abenteuer entziehen können und etwas riskieren wollen, um voranzukommen, als radikal, rebellisch oder sogar als naiv und unbelehrbar bezeichnet. Und wieder andere, die sämtliche Möglichkeiten analytisch betrachten und ihre Chancen auf Erfolg abwegen, als Strategen oder Intelligenzbolzen gewürdigt. Unabhängig einer kategorischen Einordnung haben diese Menschen einen enormen Vorteil gegenüber jenen, die sich an den Weggabelungen des Lebens niederlassen und sich wohlmöglich an den Fehlschlägen Wagemutiger ergötzen: Sie sind entscheidungsfreudig! Fähig Veränderungen voranzutreiben und sich begierig neuen Herausforderungen zu stellen. All jene lassen sich zu neuen Ufern treiben, für das Unerwartete, für den Fortschritt begeistern. Das Leben belohnt diejenigen, die etwas bewegen, treu zu ihren Taten stehen, sich Fehler eingestehen und vor allem vor keiner Aufgabe zurückschrecken.

Als meine Promotionsphase begann, war mir anfänglich nicht bewusst, wie viele Entscheidungen ich zu treffen hatte, welche Wege ich beschreiten oder besser umschiffen sollte. Etwas blauäugig stolperte ich in meinen neuen Lebensabschnitt hinein, lediglich von der Hoffnung getrieben, diesen erfolgreich beenden zu können bzw. mit den neu gewonnenen Erfahrungen zuversichtlich und selbstsicher in die Zukunft blicken zu dürfen. Nur langsam ließ sich eine gewisse Grundordnung in dem vor mir liegenden Entscheidungschaos erkennen. Einige Pfade führten zu neuen, unvorhersehbaren Problemstellungen, nur wenige entsprachen einem gesuchten Puzzleteil für das große Ganze und die meisten, zunächst vielversprechenesten Wege entpuppten sich letztlich als Sackgassen. Doch selbst aus den vielen Irrwegen konnten die richtigen Schlüsse gezogen werden, die mich am Ende meinem Ziel immer näher brachten. Und um es mit den Worten von *Thomas A. Edison (1847-1931)* zu sagen:

„I have not failed. I've just found 10.000 ways that won't work."

Ein unverzichtbarer Bonus, welcher mit der Wahl eines Lebensweges einhergeht, ist das Aufeinandertreffen mit den verschiedenesten Persönlichkeiten, die für frischen Wind bzw. Abwechslung sorgen und in ganz besonderen Fällen einem ein Leben lang bei weiteren Entscheidungen zur Seite stehen. Einigen von ihnen, die mich während meiner Promotionsphase und darüber hinaus begleitet haben, möchte ich im Folgenden meinen tiefsten Dank aussprechen.

Danksagung

Zunächst möchte ich meinem Förderer Herrn Prof. Dr. Manfred Sommer für all seine Unterstützung danken. Er hat mir die Chance gegeben mich frei zu entfalten, mich weiter zu entwickeln, den Glauben an mich selbst bestärkt, mich persönlich sowie beruflich gefordert und mich stets dazu ermuntert an neuen Herausforderungen zu wachsen. Ich bin froh und stolz darauf ihm in den letzten Jahren als sein Mitarbeiter assistieren, an seiner unermütlichen Begeisterung Studenten Wissen zu vermitteln teilhaben und von seinen Lebenserfahrungen lernen zu dürfen.

„*Danke Manfred, dass Du immer an mich geglaubt und mir zu jeder Zeit den Rücken gestärkt hast. Ohne Dich, wäre diese Arbeit nie entstanden bzw. vollendet worden. DANKE!*“

Einen weiteren Dank gebührt Herrn Prof. Dr. Michael Guthe aufgrund seiner *inspirierenden* Betreuung bzw. Evgenij Derzapf, der mich oftmals zur Verzweiflung gebracht und mir im gleichen Maße geholfen hat.

Des Weiteren möchte ich meinen Eltern sowie meinem Bruder den tiefsten Dank für ihre Bemühungen und für die vielen motivierenden Anregungen aussprechen.

„*Ohne Eure familäre Herzlichkeit wäre dies ebenfalls nicht möglich gewesen.*“

Und natürlich, nicht zu vergessen, einen großen Dank an all meine Freunde und Bekannten, die meinen Gemütsschwankungen bzw. Launen ausgesetzt waren, für die nötige Ablenkung sorgten und mich des Öfteren daran erinnerten, das Leben nicht ganz so ERNST zu nehmen und die Arbeit, Arbeit sein zu lassen. An dieser Stelle seien vor allem Antje Pankow sowie Steffen Hartmann bedankt, die mit der Qual des Korrekturlesens konfrontiert wurden und diese Meisterleistung überlebt haben.

„*Vielen herzlichen Dank!*“

Inhaltsverzeichnis

KAPITEL 1

Einleitung

In vielen Bereichen des Alltags ist es notwendig, vorliegende Informationen aufzuarbeiten und visuell ansprechend wiederzugeben. Je nach Kontext werden für eine korrekte Darstellung der jeweiligen Sachverhalte individuell gestaltete Lösungsansätze gefordert. Sind die zugehörigen Angaben lückenhaft, müssen entsprechende Daten entweder geschätzt, rekonstruiert oder modelliert werden. Wird eine derartige Analysephase erfolgreich absolviert, können die daraus resultierenden Datensätze zu Animations- bzw. Simulationszwecken verwendet werden. Dies gehört zu den Kernaufgaben der Computergrafik, wobei sich deren Anwendungsmöglichkeiten nicht nur auf die Film- oder Spielebranche beschränken, sondern in sämtlichen Forschungs- bzw. Wirtschaftzweigen zu finden sind.

Auch bei der Entwicklung von Frühwarnsystemen, wie sie beispielsweise in den modernsten Fahrzeugen integriert wurden, spielt die Computergrafik eine wichtige Rolle. Hier kann mithilfe von zwei Kameras die direkt vor einem Fahrzeug befindliche Umgebungssituation aufgenommen und die erfassten Bilddaten dazu genutzt werden, um Bewegungsabläufe vorhersagen zu können. Werden die aus den Aufnahmen ablesbaren Daten zu allen sichtbaren Objekten dahingehend analysiert, sodass eine automatische Berechnung zu deren Ausrichtung bzw. Beschleunigung möglich wird, kann ein Bremsvorgang ausgelöst werden, sobald eine Kollision mit einem sich nähernden Objekt bevorsteht, um diese letztlich abzuwenden. Zu diesem Zweck müssen die Tiefeninformationen bzw. die Abstände zwischen den Objekten rekonstruiert und anhand dessen potentielle Gefahren abgeschätzt werden. Andererseits können Fotografien auch zur Begutachtung von Fahrzeugschäden verwendet werden, indem unter Berücksichtigung der zum Fahrzeug gehörenden Herstellerdaten mögliche Verformungen der Karosserie nachmodelliert und mit den vorliegenden Informationen verglichen werden.

Für die beiden exemplarisch genannten Anwendungsfälle stellt die Rekonstruktion und Verarbeitung von Objekten bzw. kompletten Szenen aus zweidimensionalen Kamerabildern eine wichtige Grundvoraussetzung für die jeweilige Datenanalyse dar. Aus diesem Grund werden mit dieser Dissertation einige Möglichkeiten zur Reproduktion dreidimensionaler

Modelle oder ganzen Raumsituationen erörtert sowie die Modellierung von Objekten und die damit im Zusammenhang stehenden Vorbedingungen diskutiert. Da die Vorgehensweisen zur 3D-Rekonstruktion (siehe Kapitel 2) und der Modellierung (siehe Kapitel 3) unabhängig voneinander betrachtet werden können, wurde diese Arbeit in zwei separate Kernbereiche untergliedert, innerhalb derer neben den entsprechenden Grundlagen, die verwandten Forschungsarbeiten sowie die eigenen Entwicklungsideen vorgestellt werden. In dem abschließenden Kapitel 4 werden die wichtigsten Aspekte der eigenen Arbeit aus beiden Themengebieten nochmals zusammengefasst und eine Gesamteinschätzung abgegeben.

KAPITEL 2

3D-Rekonstruktion aus reziproken Bildpaaren

In der Computergrafik beschäftigt sich die 3D-Rekonstruktion mit der Nachbildung bzw. der Vervollständigung dreidimensionaler Objektstrukturen. Dementsprechend werden entweder komplette Modelle oder einzelne Bestandteile eines Modells mithilfe von analytischen Verfahren reproduziert.

Sollen Teile eines Modells rekonstruiert werden, liegt bereits ein dreidimensionales Polygonfragment vor, welches dazu verwendet werden kann, die fehlenden Komponenten nachzubilden. In der Regel wird hier davon ausgegangen, dass ein 3D-Modell über gewisse Symmetrieeigenschaften verfügt. Unter dieser Annahme kann eine Kopie des Fragmentes über die nicht vorhandenen Polygonanteile projiziert und solange rotiert, skaliert oder verschoben werden, bis sich diese den lokalen Begebenheiten annähert und sich ohne sichtbare Fehler in die lückenhafte Objektpassage einfügen lässt. Kann keine Übereinstimmung gefunden werden, wird in den meisten Fällen ergänzend eine mit 3D-Modellen befüllte Datenbank nach ähnlichen Strukturen durchsucht.

Im Gegensatz dazu kann bei der Rekonstruktion eines kompletten dreidimensionalen Objektes von keinem Basismodell ausgegangen werden. Abhängig von den Eingabedaten müssen hier komplexere, mathematische Zusammenhänge betrachtet und ausgewertet werden. Soll ein real existierendes Objekt digital erfasst werden, besteht entweder die Möglichkeit, dieses mittels eines Scanners zu erfassen oder aus mehreren Betrachtungswinkeln heraus zu fotografieren. Wurde das gewünschte Objekt eingescannt, liegt dieses im Anschluss in Form einer Punktwolke vor. Ziel ist es nun, die einzelnen Punkte miteinander zu verbinden, sodass ein exaktes, virtuelles Abbild konstruiert werden kann. Werden Bilder zur Rekonstruktion verwendet, müssen zunächst aus den zweidimensionalen Bilddaten die Tiefeninformationen für das Modell gewonnen werden. Dabei ist es wichtig Korrespondenzen zwischen den einzelnen Bildern aufzudecken, mit deren Hilfe die separat erfassten Objektteile fehlerfrei miteinander verknüpft werden können. Wurden die Tiefeninformationen sowie die für die Ausrichtung der Bestandteile benötigten Zusammenhänge extrahiert, kann daraus eine Punktwolke und schließlich eine dreidimensionale Objektrepräsentation generiert werden. Bei beiden Herangehensweisen besteht die Möglichkeit, dass während der

Datenerfassung Fehler gemacht wurden oder die entsprechenden Datensätze unvollständig sind. Dies führt zwangsläufig zu Lücken innerhalb des resultierenden Polygonnetzes, welche nachträglich ausgebessert werden müssen. Wurde ein dreidimensionales Objekt erfolgreich rekonstruiert, kann das Modell in einer Datenbank abgelegt und zum Beispiel für die Generierung virtueller Welten in Animationsfilmen oder in der Spieleprogrammierung verwendet werden.

Bevor im Folgenden die 3D-Rekonstruktion aus zweidimensionalen Bilddaten im Detail erörtert wird, sollen neben den relevanten Grundlagen auch die Grundprinzipien verwandter Forschungsarbeiten dargelegt werden. Des Weiteren wird am Ende dieses Kapitels auf die Möglichkeiten zur Bildkompression eingegangen.

2.1 Grundlagen

Bei der Rekonstruktion dreidimensionaler Objekte müssen viele mathematische Teilprobleme gelöst und physikalische Begebenheiten beachtet werden. So sollen in diesem Kapitel neben den Eigenschaften einer Kamera, deren Verwendungsmöglichkeiten zur Berechnung perspektivischer Projektionen erläutert, sowie die Grundlagen zur Transformation von Bilddaten aus dem Zeitbereich heraus in ihre Frequenzdarstellung, als auch die Möglichkeiten zur Auswertung von Korrespondenzen und die damit in Verbindung stehenden Ähnlichkeitsmaße vorgestellt werden. Ergänzend sollen die wesentlichen Aspekte der Graphentheorie bzw. des Min-Cut und Max-Flow Problems diskutiert werden.

2.1.1 Kameramodell und Kamerakalibrierung

Sollen mithilfe von Bildern Objektdaten erfasst und analysiert werden, ist es wichtig ein Verständnis für die mathematischen Zusammenhänge während des Abbildungsprozesses einer räumlichen Szene auf eine zweidimensionale Bildebene zu entwickeln. Zu diesem Zweck sollen an dieser Stelle die wesentlichen Charakteristika einer Kamera beschrieben, sowie die elementaren Prinzipien der Kamerakalibrierung und der perspektivischen Projektion erörtert werden.

2.1.1.1 Das Modell der idealen Lochkamera

Bereits in der Renaissance widmeten sich Wissenschaftler und vor allem Künstler wie Leonardo Da Vinci der Frage, wie der Eindruck von räumlicher Tiefe auf einer zweidimensionalen Bildfläche erzeugt werden kann. Die wesentliche Erkenntnis bestand darin, dass alle Punkte P im $\mathbb{R}^3$ über ein optisches Zentrum C auf eine Bildebene projiziert werden können. Dieses Projektionsprinzip wird als *Lochkameramodell* bezeichnet und soll durch Abbildung 2.1 (links) schematisch veranschaulicht werden.

Das optische Zentrum C, der sogenannte Brennpunkt (engl. *focal point*), befindet sich auf der Brennebene, auch als Fokalebene (engl. *focal plane*) bekannt. Der Abstand f zwischen

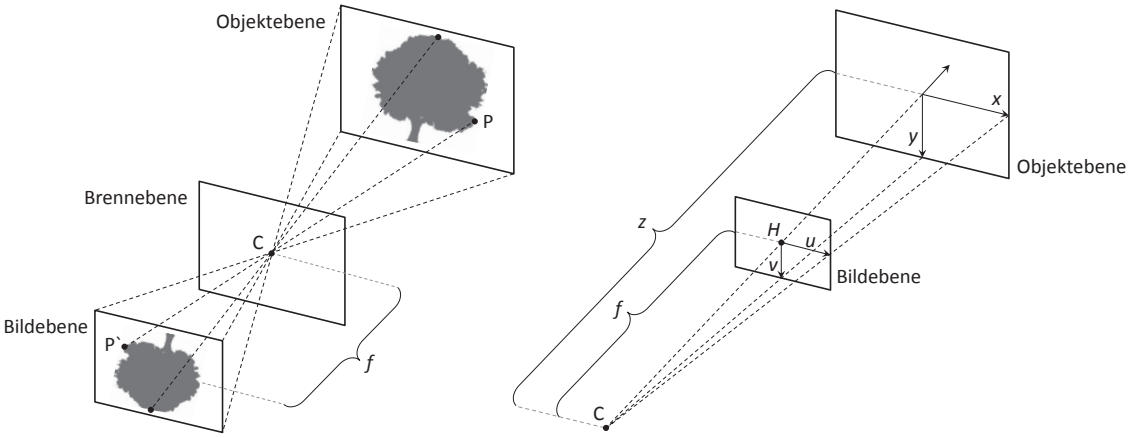


Abbildung 2.1: Projektion einer räumlichen Szene auf eine zweidimensionale Bildfläche. Links das klassische Lochkameramodell und rechts das Lochkameramodell in Positivlage. In Anlehnung an [AGD11, Sch05]

Bild- und Brennebene, wird als Brennweite (engl. *focal length*) bezeichnet. Während der Projektion wird ein Szenepunkt $P = (x, y, z)$ mittels einer Punktspiegelung an C auf die Bildebene (engl. *image plane*) übertragen, wodurch letztlich ein sowohl horizontal als auch vertikal gespiegeltes Abbild der aktuell betrachteten 3D-Szene generiert wird. Die Berechnung der Bildkoordinaten u und v lässt sich dabei direkt aus dem zweiten Strahlensatz ableiten:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{f}{z} \begin{pmatrix} x \\ y \end{pmatrix}. \tag{2.1}$$

Für die Modellierung einer zentralperspektivischen Abbildung, ohne die durch die Punktspiegelung resultierende Rotation des Bildes um 180 Grad, muss die Projektionsebene, unter Berücksichtigung von f, vor die Fokalebene verschoben werden (siehe Abbildung 2.1 rechts). In der Praxis bedeutet dies, dass ausschließlich das Vorzeichen der Koordinaten vom Projektionspunkt $P' = (u, v)$ verändert werden muss. Das *Lochkameramodell* befindet sich nunmehr in *Positivlage* [AGD11].

Mithilfe des Lochkameramodells können die bei der Abbildung einer räumlichen Szene auf eine Bildfläche auftretenden mathematischen Verhältnisse anschaulich dargestellt werden. Für eine praktische Anwendung müssen jedoch einige zusätzliche Faktoren berücksichtigt werden, deren Begrifflichkeiten im Folgenden eingeführt werden sollen [AGD11, FP12, HZ03, Sch05]. Die Erläuterungen beziehen sich dabei auf die in Abbildung 2.1 dargestellten Sachverhalte.

Optische Achse: Die optische Achse (engl. *optical axis*) verläuft senkrecht zur Bildebene, ausgehend vom Projektionszentrum C durch den Bildhauptpunkt H (engl. *principal point*) hindurch.

Bildkoordinatensystem: Im Bezug auf die Bildebene stellt das Bildkoordinatensystem ein zweidimensionales, pixelbasiertes Koordinatensystem dar, dessen Ursprung sich in der linken, oberen Ecke befindet.

Kamerakoordinatensystem: Das dreidimensionale, metrische Koordinatensystem der Kamera hat seinen Ursprung im optischen Zentrum C. Dabei verläuft die x- bzw. y-Achse parallel zur u- bzw. v-Achse des Bildkoordinatensystems und die z-Achse weist in Richtung der betrachteten Szene.

Weltkoordinatensystem: Die einzelnen Punkte einer abzulichtenden Szene werden durch Weltkoordinanten repräsentiert. Innerhalb des zugehörigen Koordinatensystems (metrisch) kann die Kamera beliebig ausgerichtet bzw. positioniert werden.

2.1.1.2 Extrinsische Parameter

Um einen Szenepunkt $P_w = (x_w, y_w, z_w)$ auf die zweidimensionale Bildebene projizieren zu können, muss P_w vom Weltkoordinatensystem in das Koordinatensystem der Kamera überführt werden. Da die Kamera beliebig im $\mathbb{R}^3$ positioniert und verdreht werden kann, wird für die Transformation der Weltkoordinaten ein Translationvektor t, welcher die Entfernung des Ursprungs vom Kamera- zu dem des Weltkoordinatensystems wiederspiegelt, sowie eine Rotationsmatrix R, welche sich durch die Ausrichtung der Kamera im $\mathbb{R}^3$ ergibt, benötigt. Für die Berechnungen können t und R zur einer Matrix kombiniert werden, sodass P_w mittels einer Matrixmultiplikation in die korrespondierende Punktdarstellung P_c umgewandelt werden kann. Für die inverse Transformation muss t komponentenweise negiert und R transponiert werden [FP12, HZ03, Sch05]:

$$P_c = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} P_w, \tag{2.2}$$

$$P_w = \begin{pmatrix} R^T & -t \\ 0 & 1 \end{pmatrix} P_c. \tag{2.3}$$

Die Komponenten für t und R werden im Zuge der Kamerakalibrierung (siehe Abschnitt 2.1.1.6) ermittelt und müssen aktualisiert werden, sobald die Kamera im Raum verschoben wird oder sich deren räumliche Orientierung ändert. Da diese Transformation nicht von den physikalischen Verhältnissen innerhalb einer Kamera abhängig ist, sondern lediglich die „*äußeren*“ Begebenheiten berücksichtigt, wird sie in der Literatur auch als *externe Transformation* bezeichnet [Sch05]. Die Werte für t bzw. R werden in diesem Zusammenhang unter dem Begriff der *extrinsischen Parameter* zusammengefasst.

2.1.1.3 Intrinsische Parameter

Nachdem der Szenepunkt P_w in das Kamerakoordinatensystem transformiert wurde, liegen die jeweiligen Punktkoordinaten von $P_c = (x_c, y_c, z_c)$ immer noch in metrischer Form vor. Um ein zweidimensionales Abbild der 3D-Szene zu erhalten, müssen diese abschließend in das pixelbasierte Koordinatensystem der Projektionsfläche transferiert werden. Zur Bestimmung der diskreten Bildkoordinaten u bzw. v, wird demnach eine horizontale sowie vertikale Skalierung mithilfe der Kamerakonstanten s_u und s_v vollzogen. Für die jeweilige Richtung, entspricht s_u bzw. s_v der Anzahl an Pixeln pro metrischer Einheit. Aufgrund der Tatsache, dass sich der Ursprung des Bildkoordinatensystems im Gegensatz zu dem der Kamera nicht im Zentrum der Bildebene befindet (siehe Abschnitt 2.1.1.1), muss der Bildhauptpunkt $H = (h_x, h_y)$ bei den Berechnungen mit einbezogen werden. Demzufolge lässt sich Formel 2.1, unter Berücksichtigung der Brennweite f, folgendermaßen modifizieren [AGD11]:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} h_x \\ h_y \end{pmatrix} + \frac{f}{z_c} \begin{pmatrix} s_u x_c \\ s_v y_c \end{pmatrix}. \tag{2.4}$$

Idealerweise kann dieser Sachverhalt auch als Matrixmultiplikation, unter Verwendung homogener Koordinaten, dargestellt werden [FP12, HZ03, Sch05]:

$$\begin{pmatrix} u\ z_c \\ v\ z_c \\ z_c \end{pmatrix} = \begin{pmatrix} f\ s_u & 0 & h_x & 0 \\ 0 & f\ s_v & h_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix}. \tag{2.5}$$

Die zugehörige inverse Transformation ist dabei nicht eindeutig definiert. Ausgehend vom Bildpunkt $P' = (u, v)$ können für z_c verschiedene Werte angenommen werden. Dies resultiert aus dem Umstand, dass alle Punkte P_w einer dreidimensionalen Szene, welche auf einer Geraden durch das Projektionszentrum C gelegen sind, wegen der Projektion in die Bildebene auf P' abgebildet werden.

Die für die Umwandlung der Punktkoordinaten vom metrischen in das pixelbasierte Koordinatensystem notwendigen Parameter werden zu den *intrinsischen Koeffizienten* gezählt. Hinzu kommen die Parameter zur Beseitigung von Bildverzerrungen, welche durch die Krümmung der in aktuellen Digitalkameras verbauten Linsen entstehen. Die Auswirkungen von optischen Verzerrungen auf die Berechnung der Bildkoordinaten werden im folgenden Abschnitt diskutiert. Da die oben geschilderte Umformung lediglich auf die der Kamera zugrunde liegenden physikalischen Eigenschaften beruht, wird diese in der einschlägigen Literatur auch als *interne Transformation* angegeben [FP12, Sch05].

2.1.1.4 Optische Verzerrungen

Moderne Kameras entsprechen nicht dem Modell der idealen Lochkamera, da aufgrund der Linsenkrümmerung in den Objektiven eine radiale sowie tangentiale Verzerrung eines projizierten Szenepunktes hervorgerufen wird. Abhängig von der Brennweite f sind im resultierenden Kamerabild vor allem die Auswirkungen der radialen Verzerrung, durch welche der Abstand eines Bildpunktes zum optischen Zentrum hin skaliert wird, erkennbar (siehe Abbildung 2.2). Die Richtung des jeweiligen Vektors zwischen Projektionszentrum und Bildpunkt ist davon nicht betroffen.

Abbildung 2.2: Beispiel für ein Kamerabild mit (links) und ohne Linsenverzerrung (rechts). Links sind die Auswirkungen der radialen Verzerrung deutlich zu erkennen.

Für eine entsprechende Korrektur bedarf es daher einer weiteren, nichtlinearen Transformation, welche direkt nach dem Transfer eines Szenepunktes P_w in das Koordinatensystem der Kamera, erfolgt. In Hinblick auf die Projektion des aus der Formel 2.2 resultierenden Punktes $P_c = (x_c, y_c, z_c)$ in die zweidimensionale Bildebene, muss dieser in die Koordinatendarstellung $P_d = (x_d, y_d)$ überführt werden, wobei $x_d = x_c/z_c$ und $y_d = y_c/z_c$ gilt. Seien die Parameter κ_1, κ_2 für die radiale und η_1, η_2 für die tangentiale Linsenverzerrung definiert, so können die entsprechenden Korrekturterme zu

$$\begin{pmatrix} u_d \\ v_d \end{pmatrix} = \rho \begin{pmatrix} x_d \\ y_d \end{pmatrix} + \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}, \; mit \tag{2.6}$$

$$\rho = 1 + \kappa_1 \, r^2 + \kappa_2 \, r^4, \tag{2.7}$$

$$\lambda_1 = 2 \, \eta_1 \, x_d \, y_d + \eta_2 (r^2 + 2 \, x_d^2), \tag{2.8}$$

$$\lambda_2 = \eta_1 (r^2 + 2 \, y_d^2) + 2 \, \eta_2 \, x_d \, y_d, \tag{2.9}$$

$$r = \sqrt{x_d^2 + y_d^2} \tag{2.10}$$

kombiniert und zur Berechnung der entzerrten Bildkoordinaten u_d bzw. v_d herangezogen werden. Unter Berücksichtigung aller *intrinsischen Koeffizienten* lässt sich der Bildpunkt $P' = (u, v)$ abschließend folgendermaßen ermitteln [AGD11, Sch05]:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \; s_u \; u_d + h_x \\ f \; s_v \; v_d + h_y \end{pmatrix}. \tag{2.11}$$

Ergänzend kann die Transformation eines Punktes vom Kamera- in das Bildkoordinatensystem inklusive der Entzerrung der Koordinaten, auch durch folgende Matrixmultiplikation repräsentiert werden:

$$\begin{pmatrix} P' \\ 1 \end{pmatrix} = A \begin{pmatrix} P_d \\ 1 \end{pmatrix}, \tag{2.12}$$

wobei A definiert ist durch:

$$A = \begin{pmatrix} \rho\ f\ s_u & 0 & \lambda_1 + h_x \\ 0 & \rho\ f\ s_v & \lambda_2 + h_y \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.13}$$

Die hier geschilderte Herangehensweise zur Korrektur der radialen und tangentialen Verzerrung ist auf die Projektion einer räumlichen Szene in ein zweidimensionales Kamerabild beschränkt. Um ein vorgegebenes Kamerabild nachträglich zu entzerren, müssen für entsprechende Modifikationen zu jedem Punkt im entzerrten Bild die korrespondierenden Koordinaten u_d bzw. v_d aus dem verzerrten Original interpoliert werden. Dies führt im Allgemeinen zu einem Informationsverlust, der sich negativ auf Folgeberechnungen auswirken kann.

2.1.1.5 Perspektivische Projektionen

In den vorangegangenen Abschnitten wurden die Prozesse zur Abbildung eines in Weltkoordinaten gegebenen Szenepunktes $P_w = (z_w, y_w, z_w)$ auf den Bildpunkt $P' = (u, v)$ erläutert. Demnach kann die Approximation einer perspektivischen Projektion in Abhängigkeit der *intrinsischen* und *extrinsischen Koeffizienten* einer Kamera beschrieben werden. In diesem Abschnitt soll zusammenfassend auf die mathematischen Beziehungen zwischen den einzelnen Projektionsschritten eingegangen werden. Dabei soll die orthographische bzw. parallele Projektion den Ausgangspunkt der Betrachtungen bilden.

Bei der Parallelprojektion spielt die Tiefe eines Raumpunktes keine Rolle, wodurch diese vernachlässigt werden kann. Demzufolge gilt $u = z_w$ bzw. $v = y_w$. Unter Berücksichtigung der *extrinsischen Koeffizienten* R und t lässt sich somit folgender mathematischer Zusammenhang in Verbindung mit einer normierten Projektionsmatrix formulieren:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}. \tag{2.14}$$

Für die Umwandlung der einzelnen Koordinaten vom metrischen in das pixelbasierte Koordinatensystem der Bildebene muss eine entsprechende Umrechnung in Abhängigkeit der Brennweite f und der Skalierungsfaktoren s_u bzw. s_v vollzogen werden. Ergänzend ist eine

Verschiebung der Koordinaten unter Verwendung des Bildhauptpunktes $H = (h_x, h_y)$ notwendig, da sich das Zentrum des Bildkoordinatensystems nicht in der Mitte der Bildebene befindet. Werden die aufgrund der Linsenkrümmung auftretenden optischen Verzerrungen mit berücksichtigt, sollten die zugehörigen Parameter (ρ, λ_1 und λ_2) ebenfalls in der Projektionsmatrix erscheinen. Um den Übergang von der parallelen zur perspektivischen Projektion korrekt beschreiben zu können, muss zusätzlich die Tiefe z_c beachtet werden, welche sich aus der Transformation von P_w in das Kamerakoordinatensystem ergibt. Zur Approximation einer allgemeinen affinen Projektion sollte schließlich noch der Scherungsparameter s eingeführt werden. Dies resultiert in folgenden Modifikationen der Formel 2.14:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \rho\ f\ s_u/z_c & s & 0 & \lambda_1 + h_x \\ 0 & \rho\ f\ s_v/z_c & 0 & \lambda_2 + h_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}. \tag{2.15}$$

Wird die Matrixmultiplikation in Formel 2.15 ausgeführt, so lässt sich folgender allgemeiner Sachverhalt konstruieren [AGD11, FP12, HZ03, Sch05]:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}. \tag{2.16}$$

Für die Projektion einer räumlichen Szene auf eine zweidimensionale Bildfläche ergeben sich somit acht Freiheitsgrade, welche sich in Abhängigkeit der *in-* bzw. *extrinsischen Kameraparameter* bestimmen lassen.

2.1.1.6 Kamerakalibrierung

Die Aufgabe der Kamerakalibrierung besteht darin, die für die Projektion einer 3D-Szene auf eine 2D-Bildfläche benötigten *intrinsischen* sowie *extrinsischen Parameter* abzuschätzen. Um die Schätzung durchführen zu können, werden sogenannte *Kalibrierungskörper* oder *Targets* in die 3D-Szene integriert. In den meisten Fällen handelt es sich hierbei um Objekte, auf denen ein Schachbrettmuster oder verschiedene Messpunkte aufgebracht wurden. Da deren Ausmaße bekannt sind, können jedem Messpunkt die exakten 3D-Koordinaten zugeordnet und aufgrund der Zuordnung Korrespondenzen zum entsprechenden Kamerabild aufgedeckt werden. Es wird also in den Kamerabildern nach bekannten Strukturen gesucht und anhand der vorliegenden Relationen die Berechnung der Kameraparameter vollzogen. Die Suche nach den jeweiligen Strukturen kann mit geeigneten Verfahren aus der Bildverarbeitung durchgeführt werden, auf deren Vorgehensweise an dieser Stelle jedoch nicht näher eingegangen wird.

Sind zu einer endlichen Menge an Messpunkten M_i, mit $(0 \leq i < N)$, die zu den entsprechenden 3D-Weltkoordinaten $P_{w_i} = (x_{w_i}, y_{w_i}, z_{w_i})$ korrespondierenden Bildpunkte $P'_i = (u_i, v_i)$ gefunden worden, kann mithilfe der Formel 2.15 bzw. 2.16 ein lineares Gleichungssystem aufgestellt werden, dessen Lösung zu den relevanten Kameraparametern führt. Werden N Messpunkte verwendet, beinhaltet das System insgesamt $2N$ Gleichungen (eine je Bildkoordinate). Um eine möglichst genaue Schätzung der gesuchten Koeffizienten zu erhalten, sollten nichtlineare Optimierungsstrategien zur Lösung des Gleichungssystems verwendet werden, wie z.B. das Gauss-Newton-Verfahren oder der Levenberg-Marquardt-Algorithmus. Insbesondere sind derartige Optimierungen von Interesse, wenn die nichtlinearen Verzerrungen berücksichtigt werden sollen. Für eine detaillierte Erläuterung der Lösungsansätze sei auf die einschlägige Literatur verwiesen [PTVF07, SW03c].

2.1.2 Epipolargeometrie

Für die Rekonstruktion eines dreidimensionalen Objektes aus Kamerabildern müssen neben den mathematischen Zusammenhängen innerhalb einer Kamera auch die geometrischen Verhältnisse zwischen zwei verschiedenen Bildern I_1 und I_2 betrachtet werden. Eine derartige Betrachtung ist nötig, um die relevanten Tiefeninformationen für einen Szenepunkt P_w rekonstruieren zu können. Die Tiefeninterpolation für P_w kann mithilfe seiner in I_1 bzw. I_2 befindlichen Projektionen P'_1 bzw. P'_2 realisiert werden, indem deren Lagebeziehung zueinander analysiert wird. Entsprechende Grundlagen sollen in diesem Kapitel vermittelt werden.

Zwei Kameras, die dazu dienen ein 3D-Objekt abzulichten und dafür im Raum positioniert werden, bilden ein Stereosystem. Dabei wird zwischen einem *achsenparallelen* und *konvergenten Stereosystem* unterschieden. Für Ersteres werden die Kameras anhand der optischen Achsen parallel zueinander angeordnet, also lediglich in der Horizontalen verschoben und nicht gegeneinander verdreht. Werden die Kameras hingegen auf ein zu reproduzierendes Objekt bzw. auf einen Konvergenzpunkt hin ausgerichtet, wird dies als *konvergentes Stereosystem* bezeichnet. Da praktische Anwendungen üblicherweise auf dem konvergenten Aufbau beruhen, werden die entsprechenden geometrischen Verhältnisse in der Literatur unter dem Begriff der *allgemeinen Stereogeometrie* eingeführt [Sch05].

Die wichtigsten Aspekte der *allgemeinen Stereogeometrie*, auch als *Epipolargeometrie* bekannt, werden in Abbildung 2.3 schematisch veranschaulicht. Die Gerade zwischen den optischen Zentren C_1 und C_2, welche die beiden Kameras repräsentieren, wird *Basislinie* und deren Schnittpunkte e_1 bzw. e_2 mit den Bildebenen werden *Epipole* genannt. In Verbindung mit dem Szenepunkt P_w wird durch die beiden Projektionszentren eine Ebene, die *Epipolarebene*, aufgespannt, welche die beiden Bildebenen schneidet. Die daraus resultierenden Schnittgeraden l_1 und l_2 stellen die *Epipolarlinien* dar, auf welchen sich sowohl die *Epipole* als auch die Projektionen P'_1 bzw. P'_2 des Szenepunktes P_w befinden.

Wird das Weltkoordinatensystem in das optische Zentrum der ersten Kamera transferiert,

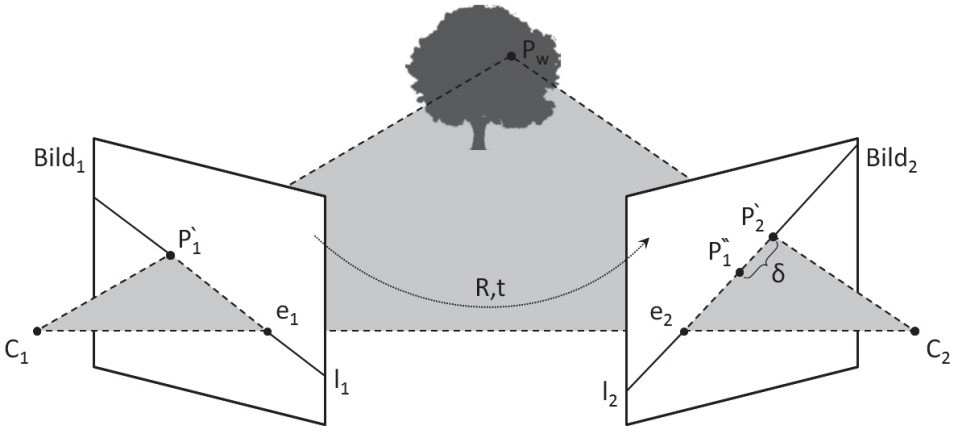


Abbildung 2.3: Schematische Darstellung zur allgemeinen Epipolargeometrie.

können mittels einer orthogonalen Rotationsmatrix R und einem Translationsvektor t alle Punkte P_{C_1} aus der ersten Bildebene in die zweite überführt werden. Formal kann dieser Sachverhalt mithilfe von

$$P_{C_2} = R\ P_{C_1} + t \tag{2.17}$$

dargelegt und für die Abschätzung der Tiefeninformationen ausgenutzt werden [HZ03].

Hilfreich ist in diesem Sinne auch folgender Sachverhalt: Wird P_w im Raum bewegt, ändert sich der Verlauf der Projektionsstrahlen zwischen dem Szenepunkt und den optischen Zentren. Dadurch wird P_w an unterschiedlichen Positionen in die beiden Bildebenen projiziert. Da die Abbildungen P_1' bzw. P_2' den Schnittpunkt eines Projektionsstrahls mit einer der Bildebenen beschreiben und Teil der *Epipolarebene* sind, können sich diese nur auf den entsprechenden *Epipolarlinien* befinden. Dies hat zur Folge, dass im Falle einer Transformation von P_1' in die zweite Bildebene der resultierende Punkt P_1'' auf der *Epipolarlinie* l_2 liegen muss. Mit anderen Worten entspricht die *Epipolarlinie* l_2 also einer in die zweite Bildebene durchgeführten Projektion der Geraden zwischen P_w und C_1.

Mithilfe des Abstands zwischen den Punkten P_1'' und P_2', auch als Disparität δ bezeichnet, können Rückschlüsse auf den Tiefenwert von P_w gezogen werden. Die Disparität stellt dabei den relativen Versatz zwischen den Projektionen P_1' und P_2' eines Szenepunktes P_w dar, wobei für die im Unendlichen gelegenen Punkte P_w die Disparität gegen Null konvergiert [HZ03, Sch05].

In praktischen Anwendungen werden oftmals zu den Kamerabildern eines *konvergenten Stereosystems* die Ansichten für das korrespondierende, *achsenparallele System* generiert. Die sogenannte *Rektifizierung* eines Bildpaares hat zur Folge, dass nach einer linearen Transformation in den parallelen Abbildern die *Epipolarlinien* l_1 bzw. l_2 immer horizontal zueinander verlaufen und zur Schätzung der Disparität δ somit lediglich die u-Koordinaten der Bildpunkte berücksichtigt werden müssen. Die wesentliche Bedingung für diesen Vorgang ist, dass die optischen Zentren der Kameras vor und nach der Transformation identisch

sind. Da das Ziel der *Rektifizierung* darin besteht die Epipolarlinien parallel auszurichten, können aufgrund der Transformation starke Verzerrungen der Objektansichten erzeugt und damit die ursprünglichen geometrischen Verhältnisse verfälscht werden. Der Grad der Objektverzerrungen richtet sich dabei nach der Größe des Winkels, welcher sich aus der Verdrehung der Kameras zueinander ergibt.

Insbesondere für die Rekonstruktion dreidimensionaler Objekte spielt die *Rektifizierung* eine bedeutende Rolle. Werden die Eingabebilder vor der eigentlichen Verarbeitung rektifiziert, können Korrespondenzen zwischen den Bildpunkten ohne größeren Rechenaufwand aufgedeckt und sowohl effizientere als auch schnellere Algorithmen entwickelt werden.

2.1.3 Reziproke Bildpaare

In den letzten Abschnitten wurden die mathematischen Beziehungen innerhalb eines Bildes sowie zwischen zwei Kamerabildern diskutiert. Im Folgenden soll es um die Beschaffenheit der in Bildern enthaltenen Informationen und um deren Verwendbarkeit gehen. Zunächst ist jeder Pixel innerhalb eines zweidimensionalen Bildes durch seinen Farbwert definiert. Wird zu einem Bildpunkt P_1' der korrespondierende Punkt P_2' im zweiten Kamerabild gesucht, muss P_1' mit einem möglichen Kandidaten verglichen und der Grad der Abweichung bestimmt werden. Entsprechende Vergleichsoperationen basieren im Allgemeinen auf der Auswertung von Farbwerten. Fehlinterpretationen sind an dieser Stelle zu erwarten, falls die in den Kamerabildern enthaltenen Objekte unterschiedlich ausgeleuchtet wurden oder Beleuchtungseffekte wie Reflektionen und Schatten an verschiedenen Positionen auf einer Objektoberfläche vorhanden sind. In solchen Fällen ist aufgrund der abweichenden Lichtverhältnisse eine korrekte Zuordnung zwischen korrespondierenden Bildpunkten nahezu unmöglich. Die Abhängigkeit von der Beleuchtungssituation kann jedoch unter Verwendung von reziproken Bildpaaren vernachlässigt werden.

Ein Bildpaar ist reziprok, wenn nach der Aufnahme des ersten Bildes für das zweite die Kameraposition mit der der Lichtquelle vertauscht wurde. Laut *Helmholtzscher Reziprozität*

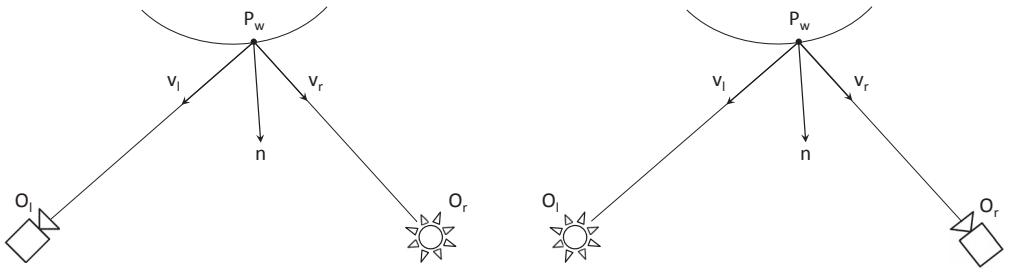


Abbildung 2.4: Schematischer Aufbau zur Erzeugung eines reziproken Bildpaares. Nach der Aufnahme des ersten Bildes (links), werden die Positionen von Kamera und Lichtquelle vertauscht, um das zweite Kamerabild aufnehmen zu können (rechts).

kann in diesem Fall das Reflektionsverhalten der Objektmaterialen in beiden Bildern als identisch angesehen werden, da die mit dem Material in Zusammenhang stehende bidirektionale Reflektanzverteilungsfunktion (engl. *bidirectional reflectance distribution function* - BRDF) unverändert bleibt [ZBK02, ZHK*03]. Abbildung 2.4 soll diese Beziehung schematisch veranschaulichen.

Entsprechen O_l bzw. O_r den Positionen der Kamera sowie der Lichtquelle und ist P_w ein Punkt auf der Objektoberfläche mit der zugehörigen Normalen $\mathbf{n}$, so kann der geschilderte Sachverhalt mithilfe folgender Formel beschrieben werden:

$$\left(e_l \frac{\mathbf{v}_l}{|O_l - P_w|^2} - e_r \frac{\mathbf{v}_r}{|O_r - P_w|^2} \right) \cdot \mathbf{n} = 0. \tag{2.18}$$

Dabei repräsentieren $\mathbf{v}_l$ bzw. $\mathbf{v}_r$ die Richtungsvektoren zwischen P_w und der Kamera bzw. Lichtquelle und e_l bzw. e_r entsprechen den Farbintensitäten, welche an den Positionen der zu P_w gehörenden Bildpunkte P_1' bzw. P_2' extrahiert werden können.

In Hinblick auf die 3D-Rekonstruktion und der damit verbundenen Korrespondenzsuche, können mittels der Kalibrierungsdaten der Kameras und einer hypothetischen Tiefe δ bis auf $\mathbf{n}$ alle Koeffizienten der Formel 2.18 bestimmt werden. Wird Formel 2.18 nach $\mathbf{n}$ aufgelöst, so kann zu jedem angenommenem δ die entsprechende Normale für P_w abgeschätzt werden. Im Zuge der Extraktion von Farbintensitäten sollte das menschliche Sehverhalten berücksichtigt werden. Da dieses empfindlicher auf Helligkeits- und weniger auf Farbunterschiede reagiert, werden im Folgenden die Grundlagen zur Transformation eines Bildes in den Frequenzbereich erörtert.

2.1.4 Fourier-Transformation

Die Fourier-Transformation (FT) ist ein mathematisches Hilfsmittel, mit welcher ein beliebiges Signal $f(t)$ aus dem Zeitbereich in den Frequenzbereich überführt werden kann. Prinzipiell wird zwischen zeitkontinuierlichen und zeitdiskreten Signalen unterschieden. Das Geräusch, welches beim Händeklatschen entsteht und die damit verbundene Schwankung des Luftdrucks ist beispielsweise ein zeitkontinuierliches Signal. Wird dieses mithilfe einer endlichen Anzahl von Abtastwerten digitalisiert, wird eine zeitdiskrete Variation des Signals generiert. Somit stellt die zeitdiskrete Darstellung eine Approximation des zeitkontinuierlichen Signales dar. Die diskrete Fourier-Transformation (engl. *discrete fourier transformation* - DFT) beruht auf der Verarbeitung derartiger Signale. Ziel einer DFT ist es, das ursprüngliche Signal durch eine Summe von Sinus- und Kosinusfunktionen mit unterschiedlichen Frequenzen und Amplituden zu repräsentieren, sodass eine Spektralanalyse des zeitbezogenen Signals möglich wird. Die theoretischen Grundlagen zur Zerlegung eines Signals in eine Summe von Grund- und Oberschwingungen wurden von Jean Baptiste

Joseph Fourier im Jahre 1807 entwickelt. Die Fourier-Reihe ist wie folgt definiert:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty}(a_k \cos(\omega_k t) + b_k \sin(\omega_k t)), \tag{2.19}$$

wobei $\omega_k = \omega_0 k$ das ganzzahlige Vielfache k einer Grundfrequenz $\omega_0 = \frac{2\pi}{T}$ mit einer Schwingungsdauer T beschreibt. In der Praxis wird die Berechnung der Fourier-Reihe nach einer endlichen Anzahl von k Gliedern abgebrochen, was zu einer Näherung des ursprünglichen Signals durch ein trigonometrisches Polynom führt. Mithilfe der Fourier-Koeffizienten a_k bzw. b_k können die Amplituden der Frequenzen ω_k gewichtet werden. Durch den Parameter a_0 wird der Offset der Funktion festgelegt, welcher die Symmetrie des Signals in Hinblick auf die Zeitachse darlegt [Bar04, Bän02, Mal98].

Zur Bestimmung der für die Signalzerlegung notwendigen Fourier-Koeffizienten kann der sogenannte *Cooley-Tukey*-Algorithmus bzw. die schnelle Fourier-Transformation (engl. *fast fourier transformation* - FFT) verwendet werden, wobei wesentliche Teile der bereits ermittelten Koeffizienten für die Berechnung anderer genutzt werden [CT65].

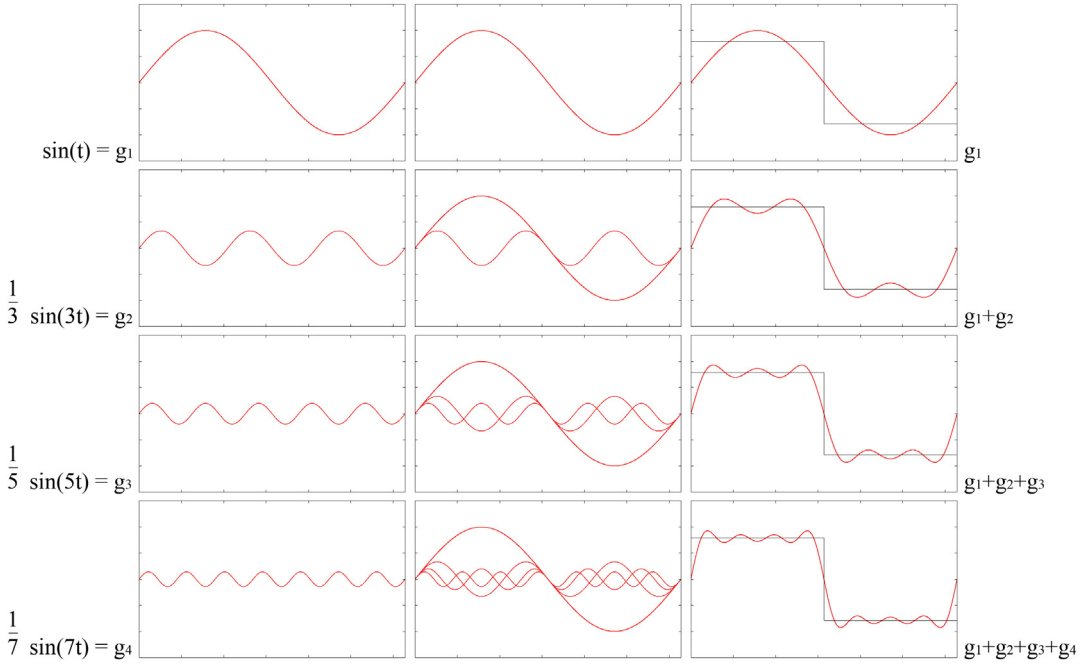


Abbildung 2.5: Die Fourieranalyse zu einer Stufenfunktion. Die Funktion kann durch die Summe der einzelnen Grund- und Oberschwingungen (links) angenähert werden (rechts). In der Mitte wird das Verhältnis der Fourierkomponenten zueinander verdeutlicht.

In Abbildung 2.5 wird die Zerlegung einer Stufenfunktion in ihre Fourierkomponenten veranschaulicht. Links sind die einzelnen Schwingungen zu sehen, deren Verhältnis zueinander in der Mitte und rechts das entsprechend approximierte Ergebnis bzw. die Summe der Sinusfunktionen. Mit zunehmender Anzahl verwendeter Fourierkomponenten steigt die Genauigkeit der Nachbildung.

2.1.5 Wavelet-Transformation

Ein Anwendungsgebiet für die Fourier-Transformation ist die Approximation von Signalen zum Zwecke der Kompression. Da die Grundfunktionen der Fourier-Transformation (Sinus, Kosinus) im Zeitbereich unendlich schwingen, lässt sich zeigen, dass die Kompression eines zeitabhängigen Signales nicht immer möglich ist und die Fourier-Transformation somit an ihre Grenzen gelangt.

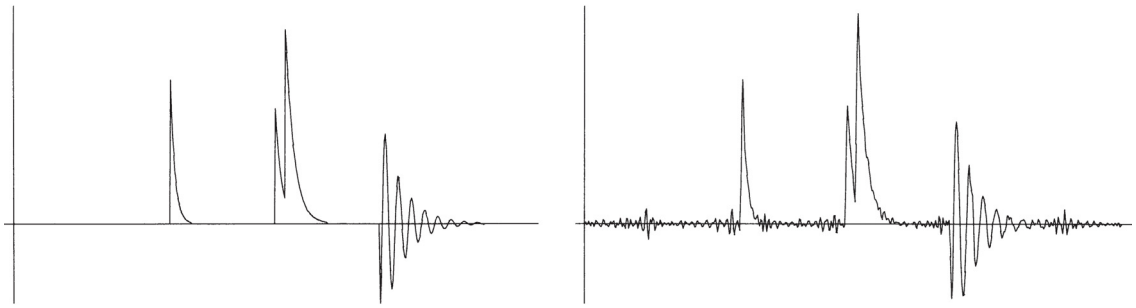


Abbildung 2.6: Die zeitliche Signalausprägung einer Funktion f (links) und deren Rekonstruktion mittels der 100 betragsgrößten Fourierkoeffizienten [Bän02].

Beispielhaft soll die in Abbildung 2.6 (links) dargestellte Funktion f möglichst genau reproduziert werden. Problematisch werden dabei die Übergänge zwischen den Intervallen, in welchen ein bzw. kein Ausschlag des Signals zu verzeichnen ist. Derartige Sprünge im Signal lassen sich nur sehr schwer mithilfe einer Sinus- bzw. Kosinusfolge imitieren. Für eine akzeptable Rekonstruktion des Signals f (Abbildung 2.6 rechts) wäre eine Vielzahl von kleineren Koeffizienten notwendig, was sich in schlechten Kompressionsraten auswirkt und bei der späteren Datenverarbeitung zu Fehlinterpretationen führen kann.

In diesem Zusammenhang können Waveletfunktionen Abhilfe schaffen, da diese aufgrund der Eigenschaft der Lokalität lediglich in einem begrentzen Intervall um die Abszisse schwingen. Außerhalb des Intervalls konvergieren diese gegen Null, wodurch eine Waveletfunktion in der Regel die Form einer dahinschwindenen Welle (franz. *ondelette* = engl. *wavelet*) annimmt. Die Güte der Signalapproximation ist dabei abhängig von der Ausdehnung und der Form des verwendeten Wavelets. Um das lokale Verhalten einer Funktion f im Zeitbereich anzunähern, sollte also ein Wavelet in Betracht gezogen werden, welches über ein geeignetes, begrenztes Intervall an von Null verschiedenen Werten bzw. über eine endliche Trägerbreite (engl. *finite width of support*) verfügt [Bän02, Mal98].

Jedes Wavelet ist durch eine charakteristische Basisfunktion ψ (auch als Mutter-Wavelet bezeichnet) definiert und kann über eine Skalierungsfunktion φ (Vater-Wavelet) gestaucht

oder gestreckt werden. Die Funktionen sind dabei von folgender Form:

$$\varphi(x) = \sum_{k\in\mathbb{Z}} a_k \, \varphi(2x-k) \tag{2.20}$$

$$\psi(x) = \sum_{k\in\mathbb{Z}} b_k \, \varphi(2x-k), \tag{2.21}$$

wobei die Werte a_k eine reellwertige Skalierungsfolge mit $\sum_{k\in\mathbb{Z}}(-1)^k a_k = 0$ repräsentieren und für die Wavelet-Folge $b_k = (-1)^k a_{1-2k}$ gilt [Dau92]. Beispielhaft ist in Abbildung 2.7 neben dem Cohen-Daubechies-Feauveau 5/3 Wavelet (CDF 5/3), das CDF 9/7 Wavelet inklusive der zugehörigen Basis- bzw. Skalierungsfunktionen abgebildet.

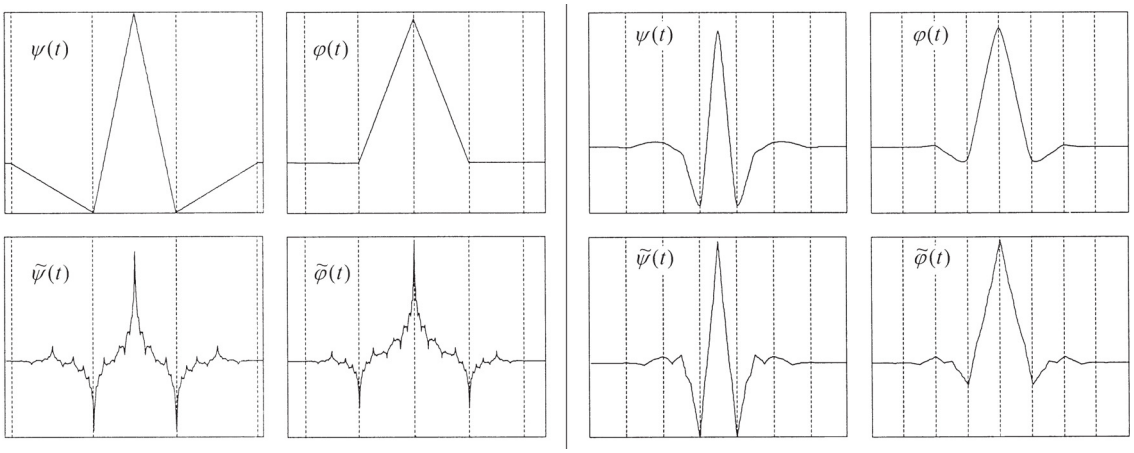


Abbildung 2.7: Das Cohen-Daubechies-Feauveau 5/3 Wavelet (CDF 5/3) mit den entsprechenden Wavelet-Funktionen für die Analyse- bzw. Synthesephase (ψ bzw. $\widetilde{\psi}$) und den zugehörigen Skalierungsfunktionen φ bzw. $\widetilde{\varphi}$ (links). Rechts die jeweiligen Funktionen für das CDF 9/7 Wavelet [TM01].

Die Wavelet-Transformation selbst, welche in erster Linie in die kontinuierliche bzw. diskrete Wavelet-Transformation unterteilt wird, setzt sich aus der Wavelet-Analyse (Übergang von der Zeit- in die Spektraldarstellung) und der Wavelet-Synthese (inverse Transformation zurück in den Zeitbereich) zusammen. Die kontinuierliche Wavelet-Transformation (engl. *continuous wavelet transformation* - CWT) wird hauptsächlich in der Mathematik und Datenanalyse angewendet, wohingegen die diskrete Wavelet-Transformation (engl. *discrete wavelet transformation* - DWT) eher im Bereich der Datenkompression bzw. Signalverarbeitung zu finden ist.

Bei der Bildverarbeitung mithilfe der diskreten Wavelet-Transformation wird ein vorgegebenes Signal bzw. Bild sukzessive in mehrere Frequenzbänder unterteilt (Analyse), die in der Summe eine vollständige Rekonstruktion der Originaldaten ergeben (Synthese). Während der Generierung dieser diskreten Teilmengen werden mehrere Auflösungen des Bildes generiert und jede einzelne mithilfe eines Hoch- bzw. Tiefpassfilters (engl. *high-*

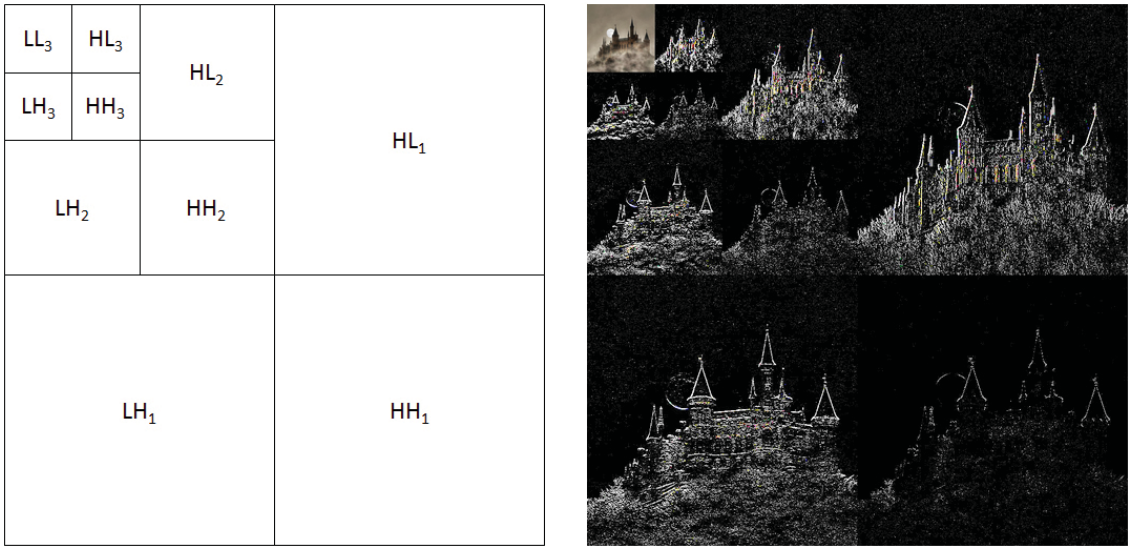


Abbildung 2.8: Schematische Darstellung einer Wavelet-Transformation und die damit im Zusammenhang stehende Bildunterteilung in mehrere Frequenzbänder. Die *LH*-, *HL*- und *HH*-Koeffizienten geben dabei die Differenzen zur nächsten Auflösungsstufe wieder und die *LL*-Koeffizienten, welche den groben Bildinformationen entsprechen, werden sukzessive in nieder- bzw. hochfrequente Bereiche unterteilt.

bzw. *low-pass filter*) in hoch- bzw. niederfrequente Anteile zerlegt. Dadurch entsteht die charakteristische Bildaufteilung in vier Bereiche, welche die für die Wiederherstellung des Originals benötigten *LL*-, *LH*-, *HL*- und *HH*-Koeffizienten beinhalten. Die hochfrequenten Schwingungsanteile beschreiben dabei die lokalen Veränderungen in der Helligkeit bzw. die Differenzen zwischen den verschiedenen Detailgraden des Bildes, wohingegen die groben Bildinformationen durch die niederfrequenten Anteile repräsentiert und in der folgenden Auflösungsstufe weiter untergliedert werden. Abbildung 2.8 soll die während einer Wavelet-Transformation vollzogene Zerlegung eines Bildes in die verschiedenen Frequenzbänder (*dyadische Dekomposition*) sowohl schematisch als auch exemplarisch veranschaulichen.

Wird für eine Anwendung der Einsatz einer Wavelet-Transformation in Betracht gezogen, wirkt sich die Wahl des Mutter-Wavelets ψ entscheidend auf die Qualität der Ergebnisse und auf die Effizienz des jeweiligen Algorithmus aus. Im Folgenden soll auf die wichtigsten Auswahlkriterien eingegangen werden [Bän02, Mal98]:

Trägerbreite: Jede Basisfunktion ψ verfügt über eine bestimmte Trägerbreite bzw. über ein begrenztes Intervall, in welchem die korrespondierenden Wavelet-Koeffizienten (endliche Folge reellwertiger Zahlen für den Tief- bzw. Hochpassfilter) ungleich Null sind. Außerhalb des Intervalls konvergiert die Funktion gegen Null. Wie bereits im Eingangsbeispiel angedeutet, wird in diesem Zusammenhang auch von der Lokalität eines Wavelets gesprochen.

Skalierungsfunktion: Mithilfe der Skalierungsfunktion φ kann die Basisfunktion ψ gestaucht oder gestreckt werden. Für die Analyse hochfrequenter Signalanteile sollte ψ gestaucht und für die Auswertung niederfrequenter Anteile gedehnt werden. Bei konstanter Skalierung kann ψ zudem entlang des Signals verschoben werden.

Regularität: Der Grad der Regularität beschreibt die Glattheit einer Basisfunktion und ist durch die Anzahl der stetigen Ableitungen von ψ definiert. Wavelets mit einer höhen Regularität wirken sich nachteilig auf den Rechenaufwand bei einer Wavelet-Transformation aus. Im Gegensatz dazu kann die Verwendung von Wavelets mit geringer Glattheit zu unerwünschten Artefakten im reproduzierten Abbild führen.

Anzahl verschwindener Momente: Für die Datenkompression ist die Anzahl der verschwindenen Momente (engl. *vanishing moments*) eines Wavelets ψ und der zugehörigen Skalierungsfunktion φ der entscheidene Faktor. Bei einer maximalen Anzahl A verschwindener Momente, verläuft die Waveletfunktion senkrecht zu jedem Polynom vom Grad höchstens (A-1), das Skalarprodukt zwischen ψ und einem Signal f ergibt also den Wert Null. Dies hat zur Folge, dass die Extraktion bestimmter Signalanteile von der Anzahl A abhängig ist. So können beispielsweise keine linearen Signale von Basisfunktionen mit nur einem verschwindenen Moment erfasst werden.

Orthogonalität: Für die genaue Wiederherstellung sowie redundanzfreie Analyse eines gegebenen Signals f sind orthogonale Wavelets vonnöten. Ein Wavelet ist orthogonal, wenn die Waveletfunktion ψ zusammen mit der Skalierungsfunktion φ eine orthogonale Basis im Hilbertraum $L^2(\mathbb{R})$ bildet (Skalarprodukt zwischen ψ und φ ist gleich Null).

Symmetrie: Für die Bildverarbeitung ist die Symmetrie (Spiegelung an der Ordinate) der Analysefunktion von großer Bedeutung, da diese eine Phasenverschiebung während der Faltung des Bildes verhindert.

2.1.6 Ähnlichkeitsmaße

Im Bereich der Bildverarbeitung werden oftmals Bildinhalte miteinander verglichen, um Analogien zwischen den betrachteten Daten aufspüren zu können. Für die Bewertung der zwischen zwei verschiedenen Bildern bestehenden Korrelation werden Ähnlichkeitsmaße definiert, die in Abhängigkeit des Anwendungsgebietes maximiert oder minimiert werden müssen. Dabei wird zwischen *nicht normalisierten* und *normalisierten Korrelatationsverfahren* unterschieden. Im Folgenden sollen einige Vertreter vorgestellt werden. Zum besseren Verständnis beschränken sich die Erläuterungen auf die reine Auswertung von Farbintensitäten zweier gegebener Graustufenbilder I_1 und I_2. Beide Bilder sind dabei von der Dimension $(M \times N)$. Die Farbintensität eines Pixels $I_1(u, v)$ bzw. $I_2(u, v)$ kann mithilfe der Bildkoordinaten u und v eindeutig bestimmt werden. Unter Berücksichtigung

dieser Vorgaben kann der Berechnung des Ähnlichkeitsgrades $K(I_1, I_2)$ folgende allgemeine Formel zugrunde gelegt werden [AGD11]:

$$K(I_1, I_2) = \sum_{u=0}^{M} \sum_{v=0}^{N} f(I_1(u, v), I_2(u,v)), \tag{2.22}$$

wobei die Funktion $f\colon \mathbb{R}\times\mathbb{R} \to \mathbb{R}$ das Analyseschema des jeweiligen Korrelationsverfahrens repräsentiert.

2.1.6.1 Summe der absoluten Differenzen (SAD)

Eines der bekanntesten nicht normalisierten Korrelationsverfahren ist die Berechnung des mittleren absoluten Fehlers bzw. der Summe der absoluten Differenzen (engl. *sum of absolute differences*). Zur Bestimmung des Ähnlichkeitsgrades zwischen I_1 und I_2 werden zu allen korrespondierenden Bildpunkten die absoluten Differenzen zwischen $I_1(u, v)$ und $I_2(u, v)$ betrachtet, aufsummiert und letztlich durch die Anzahl der Pixel dividiert [AGD11]:

$$SAD(I_1, I_2) = \frac{1}{MN} \sum_{u=0}^{M} \sum_{v=0}^{N} |I_1(u, v) - I_2(u,v)| . \tag{2.23}$$

Demnach ist die Ähnlichkeit zwischen zwei Bildern I_1 und I_2 umso höher, je kleiner die Differenz zwischen den Farbintensitäten ausfällt. Da die Auswertungsfunktion f von der Form $f(x, y) = |x - y|$ ist, entspricht dieses Ähnlichkeitsmaß der L_1-Norm [Sch05].

2.1.6.2 Summe der quadratischen Differenzen (SSD)

In der Praxis wird häufig der mittlere quadratische Fehler bzw. die Summe der quadratischen Differenzen (engl. *sum of squared differences*) als Abstandsmaß verwendet. Bevor die Summe der einzelnen Differenzen zwischen $I_1(u, v)$ und $I_2(u,v)$ gebildet und über die Anzahl der Pixel gemittelt wird, werden die Differenzbeträge quadriert:

$$SSD(I_1, I_2) = \frac{1}{MN} \sum_{u=0}^{M} \sum_{v=0}^{N} (I_1(u, v) - I_2(u,v))^2, \tag{2.24}$$

wobei $f(x, y) = (x - y)^2$ gilt [AGD11].

Basiert der Unterschied zwischen I_1 und I_2 lediglich auf einem gaußförmigen Rauschen, stellt die SSD ein verlässliches Ähnlichkeitsmaß dar. Nachteilig auf die Aussagekraft der SSD wirken sich jedoch Ausreißer aus, da diese aufgrund der Quadrierung einen stärkeren Einfluss auf das Ergebnis ausüben.

2.1.6.3 Normalisierte Kreuzkorrelation (NCC)

Für die normalisierte Kreuzkorrelation (engl. *normalized cross correlation*) kann die einfache Kreuzkorrelation (CC) mit $f(x, y) = (x \cdot y)$ als Ausgangspunkt angenommen werden:

$$CC(I_1, I_2) = \sum_{u=0}^{M} \sum_{v=0}^{N} \left(I_1(u, v) \cdot I_2(u,v)\right). \tag{2.25}$$

Problematisch wird die Verwendung der CC, wenn starke Helligkeitsunterschiede zwischen I_1 und I_2 existieren. Um die bestehenden Abhängigkeiten aufzulösen, kann jedes der Bilder über die in beiden Aufnahmen vorherrschende, durchschnittliche Helligkeit normiert werden. Dadurch beruht der Bildvergleich lediglich auf den relativen Intensitätsunterschieden zwischen korrespondierenden Bildpunkten und ist somit invariant gegenüber konstanten Divergenzen in der Helligkeit [Sch05]:

$$NCC(I_1, I_2) = \frac{CC(I_1, I_2)}{\|I_1\| \cdot \|I_2\|}, \; mit \tag{2.26}$$

$$\|I_1\| = \sqrt{\sum_{u=0}^{M} \sum_{v=0}^{N} I_1(u, v)^2} \tag{2.27}$$

$$\|I_2\| = \sqrt{\sum_{u=0}^{M} \sum_{v=0}^{N} I_2(u, v)^2} \tag{2.28}$$

Die angegebene Darstellung entspricht der L_2-Norm und verhält sich im Gegensatz zur SAD und SSD proportional zum Grad der Ähnlichkeit. Demzufolge muss für ein optimales Ergebnis der Wert der NCC maximiert werden.

2.1.7 Graph-Cut

In Hinblick auf die Reproduktion einer räumlichen Szene, werden für jeden Szenepunkt P_w verschiedene Tiefenwerte angenommen. Zu jeder hypothetischen Disparität δ werden die daraus resultierenden Projektionen P_1' und P_2' aus den Kamerabildern I_1 bzw. I_2 berechnet und anschließend deren Ähnlichkeit zueinander analysiert. Aufgrund dieser Vorgehensweise ergeben sich für einen zu rekonstruierenden Szenepunkt P_w mehrere Auswahlmöglichkeiten. Um den optimalen Tiefenkandidat zu finden, sollte der zugehörige Fehler minimiert werden. Zu diesem Zwecke können in einem dreidimensionalen Graphen die verschiedenen Tiefenvorschläge für jeden Punkt P_w angeordnet und die berechneten Ähnlichkeitswerte für die Gewichtung der Kanten verwendet werden. Dadurch ist es möglich, die Suche nach dem Optimum auf das Problem des *minimalen Schnitts* bzw. *maximalen Flusses* (engl. *max-flow min-cut theorem*) innerhalb eines Graphen zurückzuführen. Entsprechende Grundlagen sollen in diesem Abschnitt vorgestellt werden. Die Erläuterungen werden sich dabei jedoch auf die Auswertung eines zweidimensionalen Graphen beschränken.

Laut Definition wird ein gerichteter Graph G durch ein Tupel (V, E) repräsentiert, wobei V einer nicht leeren, endlichen Menge von Knoten und E der Menge an im Graphen enthaltenen Kanten entspricht. Jede Kante $(u, v) \in E$ ist durch einen Startknoten $u \in V$ bzw. Endknoten $v \in V$ eindeutig bestimmt und kann mit einer positiven Gewichtung bzw. Kapazität (engl. *capacity*) $c(u, v)$ versehen werden. Der Fluss (engl. *flow*) f ist eine Funktion $f\colon E \to \mathbb{R}_+$, welche jeder Kante $(u, v) \in E$ einen nichtnegativen Wert $f(u, v) \in \mathbb{R}_+$ zuweist und durch die Kapazitätsfunktion $c\colon E \to \mathbb{R}_+$ nach oben beschränkt ist. Für jeden Knoten $v \in V$ liegt Flusserhaltung vor, wenn der gesamte, in v über entsprechende Kanten $\delta^+(v)$ eingehende Fluss, durch seine ausgehenden Kanten $\delta^-(v)$ verteilt werden kann.

Wird ein Graph G um zwei speziell gekennzeichnete Knoten, der Quelle (engl. *source*) s und der Senke (engl. *sink*) t, für die $s \neq t$ gilt, ergänzt, so wird in der Literatur von einem Netzwerk (engl. *network*) $N = (G, c, s, t)$ gesprochen [Büs10, FF56, GT88]. Mithilfe des *maximalen Flusses* $f_{max}(s, t)$ innerhalb eines Netzwerks N können Aussagen darüber getroffen werden, wie viel Fluss f in einem Graphen G, in Abhängigkeit von der Kapazitätsfunktion c, über die einzelnen Kanten $(u, v) \in E$ zwischen der Quelle s und der Senke t maximal fließen kann. Die Kanten, durch die ein Weg zwischen s und t festlegt ist, werden unter dem Begriff des Pfades $p(s, t)$ zusammengefasst. Zur Berechnung von $f_{max}(s, t)$ werden alle möglichen Pfade $p(s, t)$ betrachtet, in denen jeder Knoten $v \in V$ maximal einmal vorkommen darf. In diesem Zusammenhang wird zusätzlich ein Residualgraph $G_f(V, E_f)$ definiert, in welchem dieselben Knoten wie in G enthalten sind und neben jeder Kante $(u, v) \in E$ eine Kante $(v, u) \in E_f$ mit der Kapazität $c_f(v, u)$ angelegt wird. Die Kapazität $c_f(v, u)$ spiegelt dabei die Differenz zwischen $c(u, v)$ und $f(u, v)$ wieder.

Werden die Knoten eines Graphen G in zwei disjunkte Teilmengen S und T aufgeteilt, so entspricht der Schnitt (engl. *cut*) $\sigma(S, T)$ der Menge an Kanten $(u, v) \in E$, welche den Übergang von S nach T beschreiben. Die Kapazität des Schnitts $c(S, T)$ ist dabei durch die Summe der zugehörigen Kantenkapazitäten festgelegt:

$$c(S, T) = \sum_{u \in S, v \in T, (u,v) \in E} c(u, v). \tag{2.29}$$

Für die Konstruktion des *minimalen Schnitts* $\sigma_{min}(S, T)$ muss eine Menge von Kanten gefunden werden, durch welche G in genau zwei zusammenhängende Teilgraphen zerlegt wird. Es existiert also keine Teilmenge in $\sigma_{min}(S, T)$, die wiederum einen Schnitt repräsentiert. Wurde ein *minimaler Schnitt* $\sigma_{min}(S, T)$ ermittelt, so ist dessen Kapazität $c_{min}(S, T)$ mit dem Betrag des *maximalen Flusses* $f_{max}(s, t)$ identisch [FF56]:

$$|f_{max}(s, t)| = c_{min}(S, T). \tag{2.30}$$

2.1.7.1 Das Verfahren von Ford und Fulkerson

Lester R. Ford und Delbert R. Fulkerson zeigten 1956, dass der *maximale Fluss* $f_{max}(s,t)$ in einem Netzwerk N dem *minimalen Schnitt* $\sigma_{min}(S,T)$ entspricht und stellten zusätzlich einen Algorithmus zur Berechnung von $f_{max}(s,t)$ vor, welcher auf dem Prinzip der Tiefensuche beruht [FF56].

Zu einem gegebenen Netzwerk $N = (G, c, s, t)$ wird zunächst der Fluss $f(u,v)$ einer Kante $(u,v) \in E$ auf Null gesetzt und der zugehörige Residualgraph $G_f = (V, E_f)$ bestimmt. Im Residualgraph G_f steht die Residualkapazität $c_f(u,v)$ einer Kante $(u,v) \in E$ für den restlichen Fluss, welcher noch von u nach v fließen kann. Die Kapazität $c_f(v,u)$ der Rückkante $(v,u) \in E_f$ stellt hingegen den Fluss dar, welcher bereits von u nach v geflossen ist. In der Summe ergeben $c_f(u,v)$ und $c_f(v,u)$ die Gesamtkapazität $c(u,v)$ einer Kante $(u,v) \in E$ im Ausgangsgraphen G.

Nach dem Initialisierungsschritt wird in G_f mittels Tiefensuche ein Pfad $p_f(s,t)$ zwischen der Quelle s und der Senke t gesucht. Wurde ein solcher Weg $p_f(s,t)$ gefunden, wird die minimale Kapazität $\gamma = c_{f_{min}}(u,v)$ aller in $p_f(s,t)$ enthaltenen Kanten (u,v) ermittelt. Anschließend wird der Fluss $f(u,v)$ sämtlicher Kanten in $p_f(s,t)$ um γ erhöht und die Kapazitäten im Residualgraph G_f entsprechend angepasst. Dieser Schritt wird solange wiederholt, bis die Tiefensuche keinen Weg zwischen s und t liefert. Dies ist beispielsweise der Fall, wenn alle von s abgehenden Kanten $(s,u) \in E$ die Residualkapazität $c_f(s,u) = 0$ aufweisen. Kanten mit der Restkapazität Null werden als *gesättigt* angesehen.

Terminiert der Algorithmus, werden alle Knoten $u \in E$, die in G_f über eine *ungesättigte* Verbindung zur Senke t verfügen, zu der Menge T und die restlichen Knoten zur Menge S zusammengefasst, woraus der *minimale Schnitt* (beinhaltet alle *gesättigten* Kanten zwischen S und T) bzw. unter Verwendung der Formel 2.30 der Betrag des *maximalen Flusses* in N resultiert.

2.1.7.2 Der Push-Relabel-Algorithmus

Mit dem im Jahre 1988 entwickelten Verfahren von Andrew V. Goldberg und Robert E. Tarjan, lässt sich der *maximale Fluss* in einem Netzwerk N wesentlich effizienter ermitteln, indem auf die zeitaufwändige Tiefensuche verzichtet wird. Im Algorithmus wird jedem Knoten mithilfe von *Push-* bzw. *Relabel*-Operationen eine bestimmte Höhe zugewiesen und somit der in einem Knoten eingehende Fluss auf seine Nachbarn verlagert. Können keine *Push-*, *Relabel*-Operationen mehr ausgeführt werden, wurde das Maximum an Fluss in der Senke t erreicht [GT88].

Der Initialisierungsschritt entspricht dem des Verfahrens von Ford und Fulkerson (siehe Abschnitt 2.1.7.1). Ergänzend wird jedem Knoten $u \in V$ im Residualgraph G_f, eines gegebenen Flussnetzwerkes $N = (G, c, s, t)$, eine Höhe $h(u)$ zugewiesen. Zu Beginn beträgt dieser Wert für alle Knoten außer s Null. Die Höhe für s ergibt sich aus der Anzahl n, der im Graphen G enthaltenen Knoten. Abschließend wird der Fluss $f(s,u)$ aller von

s abgehenden Kanten $(s,u) \in E$ auf den Wert von $c(s,u)$ gesetzt und letztlich mit der alternierenden Ausführung der *Push*- bzw. *Relabel*-Operationen begonnen.

Während einer *Push*-Operation soll der gesamte in einem Knoten u eingehende Fluss $f(\delta^+(u))$ auf seine Nachbarn verteilt werden. Dies ist nur möglich, wenn eine der von u im Residualgraph G_f abgehenden Kanten (u,v) eine Kapazität $c_f(u,v) > 0$ aufweist und für die Höhen $h(u) > h(v)$ gilt. Sind diese Bedingungen erfüllt, wird der maximale Wert für den Flusstransfer $\gamma = min(f(\delta^+(u)), c_f(u,v))$ von u nach v ermittelt und entsprechende Fluss- bzw. Kapazitätsänderungen an den beteiligten Kanten vollzogen. Dies wird solange wiederholt, bis der in u eingegangende Fluss komplett verteilt werden konnte oder die genannten Bedingungen unerfüllt blieben.

Sollte in u noch Fluss vorhanden sein und ein Nachbarknoten v mit $h(u) \leq h(v)$ existieren, muss laut Initialisierung die zugehörige Kantenkapazität $c_f(u,v) > 0$ sein. Ist dies der Fall, wird eine *Relabel*-Operation gestartet, wobei die Höhe von u auf den Wert $h(u) = h(v) + 1$ angehoben wird und dadurch der Restfluss abfließen kann. Befinden sich mehrere Knoten in der direkten Nachbarschaft von u, deren Höhe über den Wert von $h(u)$ liegt, wird der Knoten mit der minimalen Höhendifferenz für das *Relabeling* gewählt.

Sind die Bedingungen für die *Push*- bzw. *Relabel*-Operationen nicht erfüllt, terminiert der Algoritmus und der *minimale Schnitt* bzw. *maximale Fluss* kann durch die Aufteilung von V in die beiden disjunkten Teilmengen S und T bestimmt werden.

2.2 Klassifikation von Stereoanalyseverfahren

Werden die zwei Ansichten eines Stereokamerasystems (siehe Kapitel 2.1.2) analysiert, so wird dies als *Stereoanalyse* bezeichnet. Das Ziel einer derartigen Analyse besteht darin, eine eindeutige Zuordnung zwischen korrespondierenden Bildpunkten oder Merkmalen im aktuell betrachteten Bildpaar zu finden, sodass der Abstand eines auf den Aufnahmen erkennbaren Objektes zur Kamera hin abgeschätzt werden kann. Die *Korrespondenzsuche* bzw. *Stereoanalyse* zählt zu den klassischen Aufgabengebieten der digitalen Bildverarbeitung und kann in Abhängigkeit zur algorithmischen Umsetzung in zwei grundlegende Kategorien eingeteilt werden, den pixel- und merkmalsbasierten Verfahren. Eine detailliertere Einteilung existierender Ansätze wird durch Scharstein und Szeliski [SS02] bzw. Seitz et al. [SCD*06] vorgegeben.

Im Rahmen der Stereoanalyse wird für die Suche nach den korrespondierenden Bildpunkten bzw. -merkmalen und der damit im Zusammenhang stehenden Ähnlichkeitsbewertung oftmals auf das Problem der *Energieminimierung* zurückgegriffen. Dazu werden die aus den Eingabebildern extrahierten Informationen in einem Graphen angeordnet und im Anschluss ein Graph-Cut Algorithmus (siehe Kapitel 2.1.7) durchgeführt, um das optimale Resultat zu erhalten. Eine Abhandlung über entsprechend geeignete Verfahren wurde von Boykov und Kolmogorov [BK01] sowie von Dixit et al. [DKP05] publiziert.

2.2.1 Pixelbasierte Verfahren

Bei den pixelbasierten Algorithmen zur Korrespondenzanalyse werden die einzelnen Bildpunkte der beiden Kamerabilder miteinander verglichen. Da die Bildpixel jedoch lediglich hinsichtlich ihrer Farbe und Intensität voneinander unterscheidbar sind, kann es bei den Ähnlichkeitsberechnungen zu Mehrdeutigkeiten kommen. Diese können vermieden werden, indem die direkten Nachbarpixel um den jeweiligen Bildpunkt herum bei der Analyse miteinbezogen werden. Es wird also anstelle eines einzelnen Bildpunktes, ein ganzer Block inklusive der Nachbarn betrachtet. In einem solchen Fall wird auch von einem *Block-Matching*-Verfahren gesprochen.

Die Vorgehensweise bleibt jedoch dieselbe. Für jede Bildposition $P_1(x_1, y_1)$ aus der ersten Originalaufnahme wird ein Referenzblock der Größe (m, n) um P_1 herum gewählt und mit dem entsprechenden Musterblock im zweiten Bild an der Position $P_2(x_2, y_2)$ verglichen. Für einen Ähnlichkeitsabgleich wird der Musterblock ausgehend von der Position P_2 zusätzlich in verschiedene Richtungen verschoben. Unter Verwendung eines Ähnlichkeitsmaßes kann schließlich die Bildposition bestimmt werden, für welche die Ähnlichkeit zwischen dem Referenz- und Musterblock am höchsten ist. Die Differenz zur Position P_1 kann abschließend als Disparität δ angenommen werden [Jäh05, PB99, Pra01].

Für die Extraktion der Pixelinformationen kann eine Auflösungspyramide ausgenutzt werden. Hierbei liegen die Originalaufnahmen in unterschiedlichen Auflösungen vor, welche mittels einer Unterabtastung oder Skalierung konstruiert werden können. Mit der niedrigsten Auflösung beginnend, wird mithilfe eines definierten Suchbereiches die Ähnlichkeit zwischen den beiden Kamerabildern errechnet und entsprechende Werte in der nächst höheren Auflösungsstufe als Initialwerte angenommen. Der festgelegte Suchbereich sollte dabei im Verhältnis zur Bildgröße angepasst werden. Letzteres führt zu einer genaueren Tiefenbestimmung. Fehler können jedoch nicht ausgeschlossen werden und pflanzen sich über die einzelnen Auflösungsstufen hinweg fort. Der Vorteil liegt allerdings darin, dass die Korrespondenzen schneller aufgedeckt werden können. Der zusätzliche Rechenaufwand, welcher für die Generierung der Pyramide aufgebracht werden muss, sollte dabei nicht außer Acht gelassen werden [Ana89, FJ00, KLCL05].

Nach einer ähnlichen Herangehensweise werden bei voller Bildauflösung iterativ die Blockgröße sowie der Suchbereich reduziert, wodurch wie bei der Verwendung der Auflösungspyramide eine Lokalisierung der Tiefen erzielt wird. Die Probleme sind jedoch die gleichen.

Des Weiteren existiert das Prinzip des zweistufigen *Block-Matchings*, nach welchem zunächst nur die hervorstechendsten Bildmerkmale aus dem linken Kamerabild innerhalb des kompletten Bildbereiches der rechten Aufnahme gesucht und entsprechende Distanzen kalkuliert werden. Wurden die Verschiebungsvektoren ermittelt, können neben der Blockgröße auch der Suchbereich eingegrenzt und anhand der zuvor berechneten globalen Translationen die lokalen Korrespondenzen bestimmt werden [OGH*98].

Das grundlegende Problem, welches die meisten *Block-Matching*-Verfahren aufweisen, liegt in der Annahme, dass die innerhalb eines definierten Blocks befindlichen Bildpunkte über dieselbe Disparität verfügen. Diese Annahme führt an den Objekträndern zu Fehlinterpretationen, aufgrund derer die Kanten bis zu einer halben Blockgröße nach links oder nach rechts verschoben werden können. In der resultierden Tiefenkarte wirken derartige Kanten verschmiert. Um diese Artefakte zu minimieren, entwickelten Kanade und Okutomi [KO94] ein Verfahren, mit welchem die Form und Dimension der Blöcke hinsichtlich der aus den Bildern ablesbaren Charakteristik dynamisch angepasst wird. Allerdings ist dieses Vorgehen sehr rechenintensiv. Hirschmüller et al. [HIG02] lösen dieses Problem, indem sie die Blöcke nach möglichen Kantenkonstrukten untersuchen und ggf. halbieren, sodass ein Tiefensprung optimal wiedergegeben wird.

Zudem können sich die zur Korrespondenzanalyse herangezogenen Ähnlichkeitsmaße nachteilig auf den *Block-Matching*-Ansatz auswirken. Je nach verwendetem Fehlermaß, kann sich der *Pixel-Locking*-Effekt unterschiedlich stark bemerkbar machen, bei welchem die zu den Vergleichsoperationen genutzten interpolierten Werte unnatürlich oft ganzzahlig bleiben. Der Grund für diesen Effekt konnte bis dato noch nicht entdeckt werden. Es existieren jedoch Möglichkeiten die Auswirkungen dieser Problematik mithilfe einer affinen Transformation zu lindern [MLM*07, SHM06].

2.2.2 Merkmalsbasierte Verfahren

Im Gegensatz zu den pixelbasierten Ansätzen werden bei den merkmalsbasierten Verfahren über einen Vorverarbeitungsschritt besondere Bildeigenschaften wie beispielsweise Kanten, Ecken oder andere Punkt- bzw. Liniensegmente gefiltert. Im Anschluss werden anhand der gefundenen Bildmerkmale die Korrespondenzen zwischen den Kamerabildern ausgewertet. Der zusätzliche Aufwand, welcher durch das Filtern der Merkmale betrieben wird, kann sich je nach Art des eingesetzten Analyseschemas lohnen, da die Korrespondenzsuche auf die gefilterten Bildbereiche beschränkt werden kann. Allerdings ist zu bedenken, dass nur dort Korrespondenzen bestimmt werden können, wo auch entsprechende Features im Bild gefunden werden. Für eine detaillierte Übersicht über die Verfahren, welche für die Extraktion von Merkmalen aus Kamerabildern verwendet werden können, sei auf die einschlägige Literatur verwiesen [AGD11, Jäh05, PB99, Pra01, Sch05].

Eine Möglichkeit die gefilterten Bildmerkmale für die Korrespondenzanalyse zu nutzen, besteht darin diese nach Linien, Kanten, Ecken und Flächen hierarchisch einzuordnen. Nach der Gruppierung werden für sämtliche Flächen die Ähnlichkeitswerte bestimmt, bevor die Ecken und anschließend die Kanten bzw. Linien betrachtet werden. Mit dieser hierarchischen Suche können die zu berechnenden Disparitäten mit abnehmendem Rang in Abhängigkeit zu den vorherigen Resultaten immer exakter eingestuft werden [VC95]. Um die in Bildern vorhandenen Strukturen besser von einander abgrenzen zu können, existieren Überlegungen die Objektkonturen mithilfe von B-Splines zu beschreiben. Dadurch können

die Tiefen auf Subpixelebene genau abgeschätzt werden [WK03].

Weitere Ansätze beschäftigen sich mit der Segmentierung von Kamerabildern. Lee et al. [LOH08] stellten eine Variante vor, mit welcher sie zunächst das linke Kamerabild in Farbsegmente unterteilen, diese im rechten Bild suchen und die Tiefenwerte aus den sich ergebenden Positionen heraus ableiten. Die so abgeschätzten Tiefen werden in einer finalen Phase, unter Berücksichtigung der für die benachbarten Segmente berechneten Initialwerte, gewichtet und iterativ in einen Konvergenzzustand überführt.

Zu den merkmalsbasierten Verfahren werden auch diejenigen gezählt, die einen Bildpunkt anhand der Farbintensitäten ihrer direkten Nachbarn klassifizieren. Werden die vier umliegenden Pixel als heller, gleich hell oder dunkler markiert, ergeben sich maximal $3^4 = 81$ verschiedene Musterklassen, die zur Ähnlichkeitsbewertung verwendet werden können. Kommen für einen Punkt mehrere Disparitäten in Frage, wird an dieser Stelle immer die niedrigste Tiefe angenommen, um das Aufkommen von sogenannten Phantomobjekten zu vermeiden [FI96]. Verwandte Algorithmen oder Weiterentwicklungen dieser Technik beschäftigen sich mit der Reduzierung von Artefakten, nutzen feinere Intensitätsabstufungen, um die Anzahl der Musterklassen zu erhöhen, oder bedienen sich verschiedener Fehlermaße, um für einen Performanzzuwachs zu sorgen [Ste04, ZW94].

2.3 Motivation und Ziele

Die 3D-Rekonstruktion aus reziproken Bildpaaren stellt ein sehr komplexes Problem im Bereich der Computergrafik dar, welches in viele einzelne, voneinander unabhängige Teilaufgaben zerlegt werden kann. Mit der Entwicklung moderner Grafikhardware besteht nun die Möglichkeit, die zuvor geschilderten Ansätze zur Korrespondenzanalyse und die anschließende Tiefenrekonstruktion einer dreidimensionalen Objektoberfläche parallel auf den Grafikprozessoren ausführen zu können, um somit einen erheblichen Performanzanstieg zu erzielen. Zu diesem Zweck müssen die entsprechenden Algorithmen auf ihre Parallelisierbarkeit hin geprüft, Anpassungen vorgenommen sowie geeignete Datenstrukturen aufgebaut werden. Für die Parallelisierung muss darauf geachtet werden, dass der Grafikspeicher optimal genutzt und nur die relevanten Daten geladen bzw. im Speicher gehalten werden. Dabei sollte der für den Datentransfer zwischen CPU und GPU benötigte Zeitaufwand nicht unterschätzt und auf das Notwendigste reduziert werden. Zudem müssen bei der parallelen Ausführung der Algorithmen Kollisionen bei Lese- und Schreibvorgängen abgefangen werden, um Berechnungsfehler ausschließen zu können. Bei der Ähnlichkeitsbewertung sollte auf ein adäquates Fehlermaß zurückgegriffen und die Minimierung von bekannten Artefakten mithilfe von innovativen Techniken bzw. mit einer Kombination aus verschiedensten Verfahren vorangetrieben werden. Schlussendlich ist es wünschenswert über ein parallel auf der GPU lauffähiges Programm zu verfügen, welches zweidimensionale Bilddaten einliest, eine Tiefenabschätzung für die darin befindlichen Objekte vollzieht und ein dreidimensionales Abbild von der auf den Kamerabildern sichtbaren Szenerie generiert.

Zur Abgrenzung der eigenen Forschungsarbeit lassen sich folgende Kernziele definieren:

- die Entwicklung eines hochqualitativen, parallelen Verfahrens zur 3D-Rekonstruktion
- die Generierung einer effizienten und speicherschonenden Datenstruktur zur Verwaltung relevanter Daten
- die Verwendung eines adäquaten Fehlermaßes zur Minimierung von Berechnungsfehlern
- die Verarbeitung von hochauflösenden Bilddaten
- die Rekonstruktion feinerer und spiegelnder Objektstrukturen

Im Folgenden soll ein Algorithmus zur 3D-Rekonstruktion aus reziproken Bildpaaren vorgestellt werden. Die einzelnen Berechnungsschritte wurden dabei so konzipiert, dass diese parallel auf der GPU ausgeführt werden können. Da während der Umsetzung immer wieder Rückschläge in Kauf genommen werden mussten, beschränken sich die nachfolgenden Erläuterungen nicht allein auf den endgültigen Verfahrensablauf. Ergänzend werden einige Entwicklungsphasen angedeutet, wobei lediglich auf die vielversprechendsten Variationen eingegangen wird. Nach der Diskussion zur Korrespondenzanalyse werden die Elemente der Datenstruktur in einer Übersicht dargestellt (siehe Abschnitt 2.4.1.6) und ein Zwischenfazit (siehe Abschnitt 2.4.1.7) formuliert, um entsprechende Resultate bewerten zu können. Anschließend werden die Möglichkeiten beschrieben, wie aus den konstruierten Tiefenkarten dreidimensionale Polygonnetze erstellt werden können, welche Bedingungen für diesen Prozess erfüllt sein müssen und wie die jeweiligen Berechnungsschritte parallelisiert werden können (siehe Kapitel 2.4.2). Eine Gesamteinschätzung inklusive einer Auflistung der Testergebnisse wird in den Kapiteln 2.5 und 2.6 abgegeben.

Da bei der 3D-Rekonstruktion Bilddaten verarbeitet werden und neben der Parallelisierung viel Wert auf die Optimierung des Speicherbedarfs gelegt wurde, entstand im Rahmen dieser Arbeit zusätzlich eine Technik zur Bildkompression (siehe Abschnitt 2.7). Da diese jedoch in keinem direkten Zusammenhang zur 3D-Rekonstruktion steht, werden die algorithmischen Abläufe in einem separaten Kapitel erläutert.

2.4 Die 3D-Rekonstruktion (Ein Entwicklungsprozess)

Ein Foto stellt ein zweidimensionales Abbild der dreidimensionalen Welt dar. Jedes darin enthaltene Objekt kann durch die Menge der zugehörigen Bildpunkte beschrieben werden. Ein solcher Bildpunkt P ist durch seine Koordinaten $P(x, y)$ gegeben und verfügt über einen spezifischen Farbwert. Für eine korrekte Darstellung im $\mathbb{R}^3$ müssen die fehlenden z-Koordinaten der einzelnen Bildpunkte P ermitteln werden, um die notwendige Tiefenwirkung zu erhalten. Um dies zu bewerkstelligen, beruht die 3D-Rekonstruktion aus

zweidimensionalen Bilddaten auf der Suche nach korrespondieren Bildpunkten innerhalb eines gegebenen reziproken Bildpaares. Dabei wird ausgehend von einem Bild, zu jedem der Bildpunkte P, bzgl. seiner durch x und y gegebenen Position, das entsprechende Pendant P' im anderen Bild gesucht. Ist ein solches Paar (P, P') von korrespondierenden Punkten gefunden, können über die Abweichung der Positionen Rückschlüsse auf den Tiefenwert von P vollzogen werden. Bevor jedoch ein komplettes 3D-Modell bzw. eine Rundumansicht des Bildobjektes generiert werden kann, müssen verschiedene reziproke Bildpaare, welche von unterschiedlichen Betrachterpositionen aufgenommen wurden, einer solchen Korrespondenzsuche unterzogen werden. Um die zuvor gewonnen Tiefeninformationen nicht zu verlieren, werden pro Bildpaar sogenannte Tiefenkarten (engl. *depth maps*) erstellt, in welchen der ermittelte Tiefenwert eines Bildpunktes P als Grauwert hinterlegt wird. Wurden alle Bildpaare analysiert und ausgewertet, kann im Anschluss aus den erstellten Tiefenkarten ein dreidimensionales Polygonnetz erzeugt werden.

Im Folgenden sollen die einzelnen Verfahrensschritte zur 3D-Rekonstruktion von Objekten aus reziproken Bildpaaren genauer erläutert und deren Möglichkeiten zur Parallelisierung diskutiert werden. Da sich die Herangehensweise in zwei unabhängige Phasen:

- Berechnung einer Tiefenkarte
- Generierung einer dreidimensionalen Objektansicht

unterteilen lässt, werden die entsprechenden Problemstellungen und Lösungsstrategien in seperaten Abschnitten erörtert. Die Abbildung 2.9 gibt eine schematische Übersicht über die Abfolge des entwickelten Algorithmus [GG] wieder.

2.4.1 Berechnung einer Tiefenkarte

Für die Berechnung einer Tiefenkarte ist es zunächst erforderlich eine virtuelle Kamera zu definieren, aus deren Sicht die Tiefenkarte erstellt wird. Die virtuelle Kamera wird für die Platzierung und Ausrichtung des dreidimensionalen Gitters benötigt, welches die verschiedenen Tiefenebenen beinhaltet. Jede Tiefenebene steht für einen bestimmten Tiefenwert, den die verschiedenen Bildpunkte der Eingabebilder annehmen können. Die Kalibrierung der virtuellen Kamera ist abhängig von den Kalibrierungsdaten der Kameras, die zur Aufnahme der Orignialbilder verwendet wurden. Als Initialschritt werden demnach neben den Parametern, welche die Bildverzerrung durch die Linsenkrümmerung der Kameras beschreiben, auch die entsprechenden in- und extrinsischen Koeffizienten geladen. Mittels dieser Daten können die Attribute der virtuellen Kamera (siehe Abschnitt 2.4.1.1) bestimmt und die Ausrichtung des Tiefengitters (siehe Abschnitt 2.4.1.2) vollzogen werden.

Nachdem die Ausgangssituation für die Generierung einer Tiefenkarte geschaffen wurde, kann mit dem Laden eines reziproken Bildpaares begonnen werden. Optional können im Anschluss die einzelnen Bilder während einer Vorverarbeitungsphase beispielsweise in

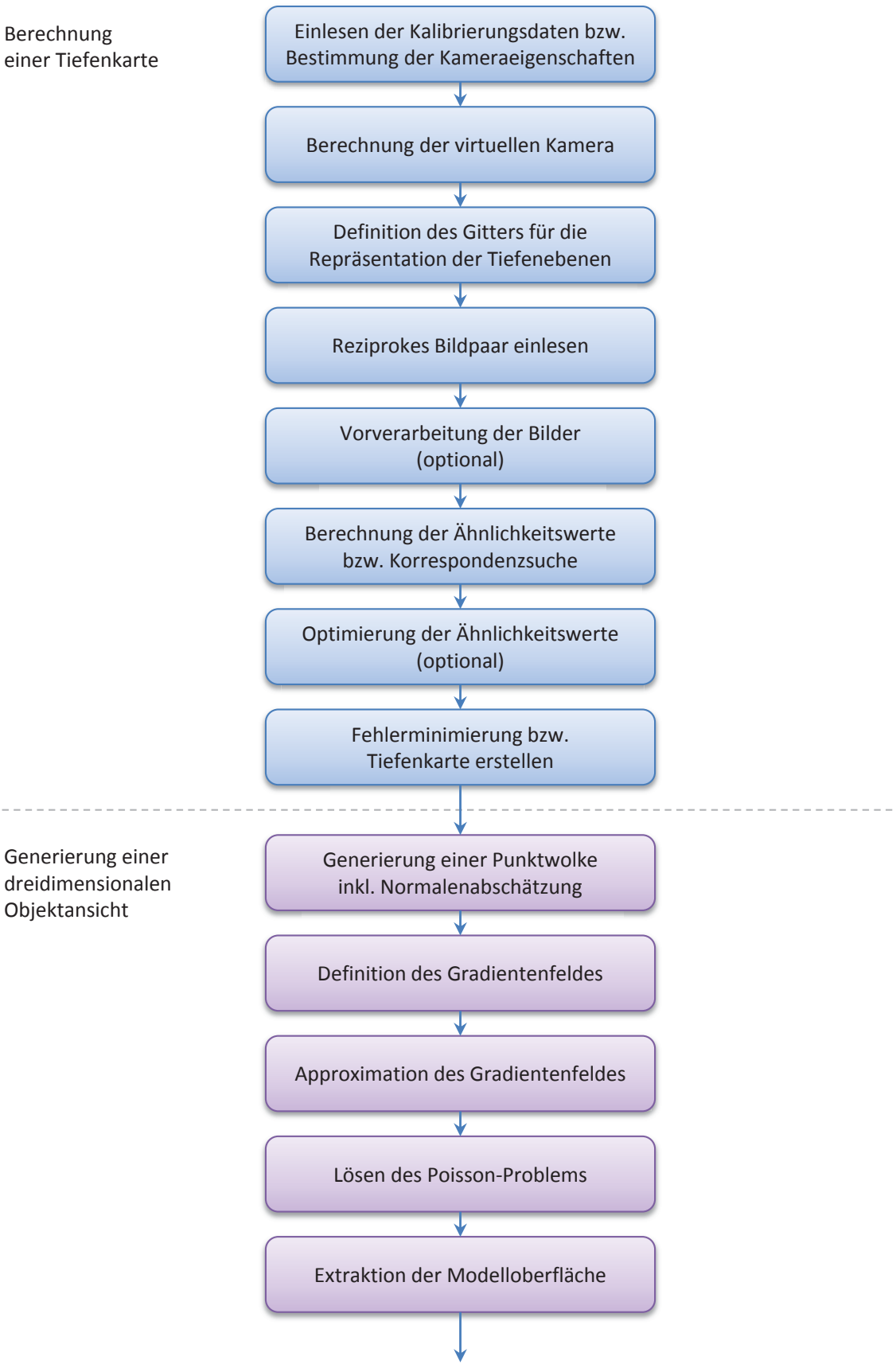


Abbildung 2.9: Schematische Darstellung für die 3D-Rekonstruktion eines Objektes aus reziproken Bildpaaren, untergliedert in zwei Hauptphasen: die Berechnung einer Tiefenkarte und deren Verwendung zur Generierung einer dreidimensionalen Objektansicht.

das lokale Koordinatensystem der virtuellen Kamera projiziert oder durch eine Wavelet-Transformation in den Frequenzbereich überführt werden. Die Vor- und Nachteile einer möglichen Vorverarbeitung bzw. deren Auswirkungen auf die Parallelisierung werden im Abschnitt 2.4.1.3 genauer diskutiert.

Bei der Korrespondenzsuche wird, ausgehend von einem angenommenen Tiefenwert, jeder einzelne Bildpunkt in die beiden Ausgangsbilder projiziert. Die an diesen Positionen befindlichen Bildinformationen (Farbwerte, Wavelet-Koeffizienten etc.) werden anhand einer Ähnlichkeitsfunktion ausgewertet und das resultierende Ähnlichkeitsmaß im zuvor definierten Gitter hinterlegt (siehe Abschnitt 2.4.1.3). Da jede im Gitter befindliche Tiefenebene einen bestimmten Tiefenwert repräsentiert, werden diese Berechnungen für jeden Bildpunkt auf sämtlichen Ebenen durchgeführt. Hinsichtlich der Parallelisierung kann die Korrespondenzsuche für jede Tiefenebene unabhängig von der anderen ausgeführt werden.

Bevor aus den ermittelten Ähnlichkeitswerten eine Tiefenkarte erzeugt werden kann, können noch einige Optimierungsschritte integriert werden, um Fehler oder Ungenauigkeiten während der Korrespondenzsuche auszugleichen. Unter anderem besteht die Möglichkeit mehrere Bildpaare zur Erstellung einer Tiefenkarte heranzuziehen. Die berechneten Ähnlichkeitswerte werden dann pro Bildpunkt aufsummiert und könnten beispielsweise im Anschluss über die Anzahl der verwendeten Bildpaare gemittelt werden. Des Weiteren wäre an dieser Stelle auch eine Wavelet-Transformation denkbar, welche auf jede Tiefenebene angewendet, zu einer Stabilisierung der darin befindlichen Informationen und damit zu einer Vermeidung von Fehlinterpretationen bei der folgenden Suche nach den optimalen Tiefenkandidaten eines Bildpunktes führen kann. Die Vor- und Nachteile der verschiedenen Optimierungsmöglichkeiten werden im Abschnitt 2.4.1.3 näher erörtert.

Abschließend wird im eingangs definierten Gitter pro Bildpunkt der Tiefenwert gesucht, welcher die höchste Ähnlichkeit bzw. die geringste Abweichung zwischen den durch die Projektion des Bildpunktes in die Originalaufnahmen ermittelten Pixelinformationen aufwies. Die so approximierte Tiefe eines Bildpunktes wird als Grauwert in der Tiefenkarte abgelegt, welche am Ende als Bild gespeichert werden kann. Da sich die Suche nicht ausschließlich auf das lokale Optimum stützen, sondern vielmehr die globalen Zusammenhänge berücksichtigen sollte, um Artefakte bzw. Fehlinterpretationen ausschließen zu können, ist es unabdingbar die Nachbarschaft der einzelnen Gitterpunkte miteinzubeziehen (siehe Abschnitt 2.4.1.5).

Im Folgenden sollen die wichtigsten Verfahrensschritte noch einmal genauer beleuchtet und am Ende dieses Kapitels zusammenfassend auf die für die Parallelisierung verwendete Datenstruktur sowie auf Zwischenergebnisse für die 3D-Rekonstruktion aus reziproken Bildpaaren (siehe Abschnitt 2.4.1.7) eingegangen werden.

2.4.1.1 Berechnung der virtuellen Kamera

Laut Definition eines reziproken Bildpaares wird bei dessen Generierung die Position der Kamera mit der der Lichtquelle vertauscht. Bei den folgenden Erläuterungen wird nicht weiter zwischen einer Kamera und einer Lichtquelle differenziert, sondern von einer linken bzw. rechten Kamera ausgegangen. Zudem wird eine Kamera nicht als physikalisches Objekt betrachtet, sondern als Kombination aus in- bzw. extrinsischen Parametern und den Verzerrungskoeffizienten angesehen.

Für ein gegebenes reziprokes Bildpaar soll eine Tiefenkarte erzeugt werden. Dazu wird eine virtuelle Kamera, aus deren Sicht die Tiefenkarte erstellt wird, zwischen den beiden Kameras platziert (siehe Abbildung 2.10). Die zugehörigen Eigenschaften können aus den in- und extrinsischen Koeffizienten der linken bzw. rechten Kamera abgeleitet werden. Die Koeffizienten für die Bildverzerrung bzw. Linsenkrümmung spielen bei der virtuellen Kamera keine Rolle, da für diese das ideale Lochkameramodell (siehe Kapitel 2.1.1.1) angenommen werden kann. Voraussetzung für die weitere Verarbeitung ist natürlich, dass das gegebene reziproke Bildpaar entsprechend entzerrt wurde (siehe Kapitel 2.1.1.4).

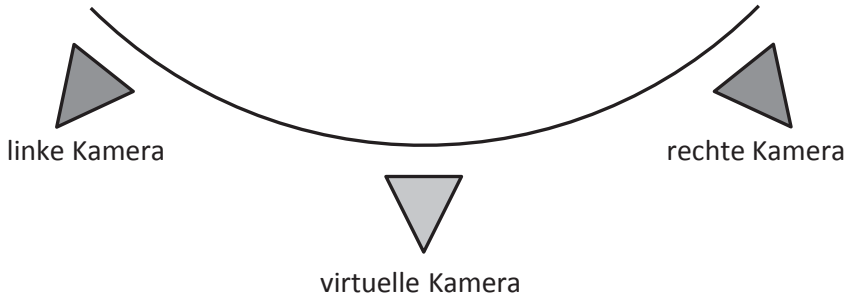


Abbildung 2.10: Schematische Darstellung zur Positionierung der virtuellen Kamera.

Da die intrinsischen Parameter (siehe Kapitel 2.1.1.3) die Abbildung zwischen dem 3D-Kamerakoordinatensystem (metrisch) und dem 2D-Bildkoordinatensystem (in Pixel) definieren bzw. die Orientierung zwischen Kamera und Bild beschreiben, müssen für die virtuelle Kamera neben den Pixelkoordinaten des Bildhauptpunktes h_x und h_y, die Werte für die Pixelskalierung in horizontaler bzw. vertikaler Richtung s_x und s_y ermittelt, sowie die Brennweite f festgelegt werden. Die gesuchten Koordinaten bzw. Skalierungsfaktoren ergeben sich aus dem arithmetischen Mittel von denen der linken und rechten Kamera. Der Bildhauptpunkt $H = (h_x, h_y)$ kann also mithilfe von

$$h_x = \frac{h_{x_{left}} + h_{x_{right}}}{2}, \qquad h_y = \frac{h_{y_{left}} + h_{y_{right}}}{2} \tag{2.31}$$

und die Faktoren der Pixelskalierung mittels

$$s_x = \frac{s_{x_{left}} + s_{x_{right}}}{2}, \qquad s_y = \frac{s_{y_{left}} + s_{y_{right}}}{2} \tag{2.32}$$

berechnet werden. Die Brennweite f sollte identisch mit denen der beiden Kameras sein und kann ohne weitere Berechnungen übernommen werden.

Die extrinsischen Eigenschaften (siehe Kapitel 2.1.1.2) beschreiben die Beziehung zwischen dem 3D-Weltkoordinatensystem und dem der Kamera. Dieser Zusammenhang wird durch eine Rotationsmatrix R und einen Translationsvektor T verdeutlicht. Ähnlich wie bei den intrinsischen Koeffizenten der virtuellen Kamera können auch R bzw. T durch das komponentenweise Bilden eines Mittelwertes bestimmt werden. Lediglich die Rotationsmatrix R muss zusätzlich orthonormalisiert werden, sodass die Spalten der Matrix orthogonal zueinander sind und ihre Norm 1 beträgt. Dies lässt sich mit folgenden Formeln beschreiben:

$$u_n = \frac{u}{\|u\|} \tag{2.33}$$

$$v_n = \frac{v'}{\|v'\|} \quad mit \quad v' = v - (u_n v)u_n \tag{2.34}$$

$$w_n = \frac{w''}{\|w''\|} \quad mit \quad w'' = w' - (v_n w')v_n \quad und \quad w' = w - (u_n w)u_n, \tag{2.35}$$

wobei u, v und w den einzelnen Spaltenvektoren, der durch die Mittelwertbildung angepassten Matrix R, entsprechen.

2.4.1.2 Gitterdefinition für die Tiefenebenen

Nachdem die virtuelle Kamera definiert wurde, kann das dreidimensionale Gitter für die verschiedenen Tiefenebenen platziert werden. Dazu muss die Anzahl der enthaltenen Tiefenebenen N und deren Dimension (Breite und Höhe) sowie der Abstand zwischen den einzelnen Ebenen festgelegt werden. Idealerweise sollte dies anhand der Auflösung der Originalbilder entschieden werden. Je größer die Auflösung der Bilder, desto größer sollte die Tiefenauflösung des 3D-Gitters und damit die Varianz in der z-Koordinate ausfallen. Gleiches gilt für die Dimension einer angelegten Tiefenebene. Das Gitter sollte so im $\mathbb{R}^3$ positioniert werden, sodass sich der zugehörige Mittelpunkt im Weltkoordinatenursprung $C(0, 0, 0)$ befindet. Dies lässt sich mithilfe des Translationsvektors der virtuellen Kamera erreichen. Eine Darstellung, welche die Positionierung und Ausrichtung eines Gitters mit N Ebenen bzgl. der verschiedenen Kamerapositionen schematisch wiederspiegelt, ist in Abbildung 2.11 zu finden.

Der in der Abbildung 2.11 als d_m bezeichnete Abstand steht für die Distanz vom Mittelpunkt C bis hin zum Rand des Gitters, sowohl in horizontaler als auch in vertikaler Richtung. Dieser sollte sich an der Bildbreite der Originalaufnahmen (*width*) orientieren und kann unter Verwendung folgender Formel berechnet werden:

$$d_m = \frac{width * w}{2}, \tag{2.36}$$

wobei w durch

$$w = \frac{dist}{0.5 * (s_x + s_y)} \tag{2.37}$$

definiert ist und das Verhältnis zwischen dem Abstand der virtuellen Kamera zum Ursprung C (als *dist* gekennzeichnet) und der Pixelbreite (über die Skalierungsfaktoren s_x und s_y der virtuellen Kamera gemittelt) wiedergibt. Mit anderen Worten beschreibt w also die metrische Breite eines Pixels, welche man aus der Entfernung *dist* wahrnehmen würde.

Mit der Berechnung von d_m lassen sich nun auch die Abstände d_i zwischen der virtuellen Kamera und den einzelnen Tiefenebenen i (mit $0 \leq i < N$) ermitteln. Die Verteilung der einzelnen Tiefenebenen kann dabei linear oder logarithmisch erfolgen. An dieser Stelle wird lediglich die lineare Variante diskutiert, wodurch sich d_i wie folgt berechnen lässt:

$$d_i = dist - d_m + \frac{2 * i * d_m}{N - 1} \ . \tag{2.38}$$

Die so berechneten Distanzen d_i repräsentierten die verschiedenen z-Koordinaten im $\mathbb{R}^3$. Die zugehörigen x- bzw. y-Koordinaten sind durch die Breite bzw. Höhe der Gitterebenen gegeben, wodurch nun jeder Punkt im dreidimensionalen Gitter definiert ist und mit der Korrespondenzsuche begonnen werden kann.

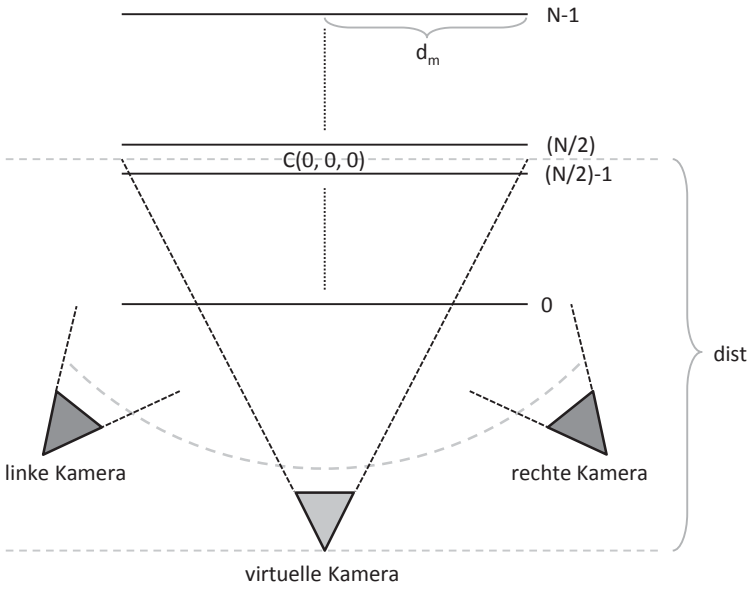


Abbildung 2.11: Die Ausrichtung eines dreidimensionalen Gitters mit N Ebenen. Die Positionierung erfolgt in Bezug auf die vorhandenen Kameras.

2.4.1.3 Berechnung der Ähnlichkeitswerte (Korrespondenzsuche)

Nachdem die virtuelle Kamera konfiguriert und das Gitter als Repräsentation der verschiedenen Tiefenebenen im $\mathbb{R}^3$ platziert bzw. ausgerichtet wurde, kann mit der Korrespondenzsuche begonnen werden. Wie die Berechnung der Ähnlichkeitswerte vollzogen, optimiert

und parallelisiert werden kann bzw. wie sich eine mögliche vorangestellte Bildvorverarbeitung auf die Ermittlung der Ähnlichkeiten und deren Umsetzung auswirkt, soll im Folgenden diskutiert werden. An dieser Stelle sei jedoch bereits darauf hingewiesen, dass als erste Optimierungsmöglichkeit die eingelesenen Bildpaare in das lokale Koordinatensystem der virtuellen Kamera überführt werden sollten. Dies hat den Vorteil, dass die Projektion der einzelnen Gitterpunkte lediglich auf die in- bzw. extrinsischen Parameter der virtuellen Kamera zurückgeführt und die Daten der beiden anderen Kameras vernachlässigt werden können. Zudem muss bei der folgenden Berechnung der Ähnlichkeitswerte nicht zwischen den verschiedenen Kameras unterschieden werden.

Im Allgemeinen lässt sich die Herangehensweise in drei Phasen unterteilen:

1. **Projektion eines Gitterpunktes in das reziproke Bildpaar**
   Jeder Punkt des Gitters muss zunächst vom pixelbasierten Koordinatensystem der virtuellen Kamera in das metrische Weltkoordinatensystem transferiert werden, bevor dieser in die beiden Originalbilder projiziert werden kann.

2. **Extraktion der Pixelinformationen**
   Mit der Projektion eines Punktes in die beiden Kamerabilder können an den entsprechenden Positionen unter Berücksichtigung der direkten Nachbarschaft die relevanten Pixelinformationen extrahiert werden. Da hierfür eine Transformation der zuvor berechneten Weltkoordinaten in die lokalen Koordinatensysteme der beiden Bildaufnahmen notwendig ist und diese keine genauen Pixelpositionen liefert, müssen die Pixelinformationen der Projektionen aus denen der direkten Nachbarschaft interpoliert werden.

3. **Ähnlichkeitsbewertung der interpolierten Pixelinformationen**
   Die entsprechenden Pixeldaten werden anhand einer Ähnlichkeitsfunktion bewertet und das sich daraus ergebene Ähnlichkeitsmaß bzw. der resultierende Grad der Abweichung wird an der zugehörigen Position im dreidimensionalen Gitter hinterlegt.

Ohne eine Vorverarbeitung der Bilder lässt sich die Auswertung der Pixelinformationen auf den einfachsten Fall reduzieren. Der naive Ansatz beruht lediglich auf dem Vergleich von Farb- bzw. RGB-Werten der einzelnen Projektionspunkte und soll daher dazu genutzt werden, um im Folgenden das grundlegende Prinzip der Korrespondenzsuche aufzuzeigen.

01.
Korrespondenzsuche
Naiver Ansatz
(ohne Vorverarbeitung der Eingabebilder)

Für die Projektion eines Gitterpunktes P_i in das reziproke Bildpaar muss dieser in das metrische Weltkoordinatensystem überführt werden. Dazu wird zunächst ein Strahl $direction_i$ vom optischen Zentrum der virtuellen Kamera in Richtung des Punktes P_i definiert. Der Abstand d_i einer im Gitter befindlichen Tiefenebene i zur virtuellen Kamera wird dabei als Tiefenwert für P_i angenommen, bzw. repräsentiert den Wert der fehlenden z-Koordinate. Der Ursprung des Weltkoordinatensystems $origin_{world}$ ergibt sich aus den

extrinsischen Koeffizienten der virtuellen Kamera bzw. durch eine Multiplikation des negativen Translationsvektors T mit der zugehörigen Rotationsmatrix R:

$$origin_{world} = -T * R \tag{2.39}$$

$$P_{i_{world}} = origin_{world} + direction_i \tag{2.40}$$

Nach der Berechnung entspricht $P_{i_{world}}$ der Darstellung von P_i in Weltkoordinaten. Während der Projektion von $P_{i_{world}}$ in eines der reziproken Bilder werden dessen Koordinaten in das jeweilige Kamerakoordinatensystem verschoben, entsprechend der intrinsischen Koeffizienten entzerrt und in pixelbasierte Koordinaten umgewandelt. Die genaue Vorgehensweise einer Punktprojektion wurde bereits im Grundlagenkapitel 2.1.1.5 erläutert, weshalb an dieser Stelle auf eine detailliertere Beschreibung verzichtet wird.

Da sich durch die vollzogene Koordinatenumwandlung keine genauen Pixelpositionen ergeben, muss der jeweilige Farbwert $col(P_{i_{l/r}})$ für die Projektionen von $P_{i_{world}}$ (P_{i_l} und P_{i_r}) aus denen der umliegenden Pixel im linken bzw. rechten Eingabebild interpoliert werden. Dazu wird jeweils eine Nachbarschaft von K Bildpunkten P_k mit $1 \leq k \leq K$ definiert, deren Farbwerte $col(P_k)$ entsprechend der Distanz zwischen P_k und $P_{i_{l/r}}$ gewichtet und aufsummiert werden. Im Anschluss wird die resultierende Summe durch die Anzahl K der betrachteten Bildpunkte gemittelt. Die Farbwertinterpolation für $col(P_{i_{l/r}})$ kann demnach als gewichteter Mittelwert folglich definiert werden:

$$col(P_{i_{l/r}}) = \frac{\sum_{k=1}^{K} col(P_k)(1 - \Delta x_k(P_k, P_{i_{l/r}}))(1 - \Delta y_k(P_k, P_{i_{l/r}}))}{K}, \tag{2.41}$$

wobei $\Delta x_k(P_k, P_{i_{l/r}})$ bzw. $\Delta y_k(P_k, P_{i_{l/r}})$ die Abweichung zwischen den Projektionen P_{i_l} bzw. P_{i_r} und dem Bildpunkt P_k in x- bzw. y-Richtung beschreibt. Mit der angegebenen Definition ist die Gewichtung des Farbwertes $col(P_k)$ eines benachbarten Pixels P_k um so höher, desto geringer die Distanz zwischen den jeweiligen Punkten ausfällt. Somit

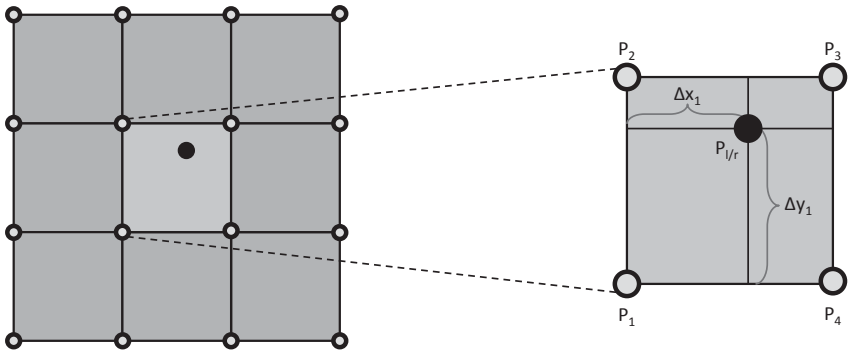


Abbildung 2.12: Farbwertinterpolation für einen projizierten Bildpunkt $P_{i_{l/r}}$ mit K=4. Die Gewichtung der Farb- bzw. RGB-Werte eines Bildpunktes P_k wird anhand der Differenzen Δx_k und Δy_k vollzogen.

ist der Farbwert $col(P_k)$ ausschlaggebend für den Farbwert der Projektionen $col(P_{i_{l/r}})$. Abbildung 2.12 zeigt einen möglichen Interpolationsfall für K=4.

Nachdem die Farbwerte für P_{i_l} und P_{i_r} interpoliert wurden, kann mit der Ähnlichkeitsanalyse fortgefahren werden. Da an dieser Stelle mehrere mögliche Alternativen zur Bewertung der Ähnlichkeit herangezogen werden können (siehe Kapitel 2.1.6), beschränken sich die folgenden Ausführungen auf die im Verfahren verwendete Normalisierte Kreuzkorrelation.

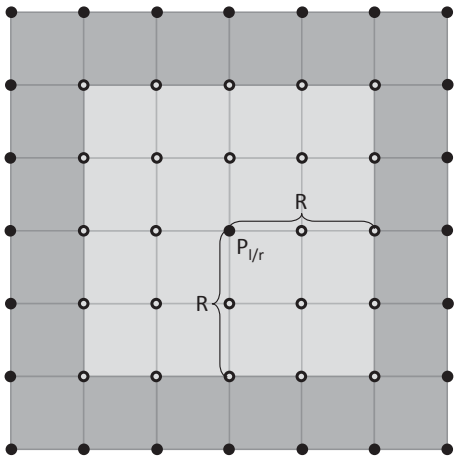


Abbildung 2.13: Die Definition einer Nachbarschaft für $P_{i_{l/r}}$ mit R=2. Die zugehörigen Pixel wurden entsprechend markiert.

Da die Bewertung der Ähnlichkeit zwischen P_{i_l} und P_{i_r} nicht allein auf lokale Eigenschaften beruht, sondern die globalen Zusammenhänge mit einbeziehen sollte, ist es wichtig, deren direkte Nachbarschaft und die zugehörigen interpolierten Farbwerte ebenfalls zu betrachten. In diesem Sinne wird über einen Radius R eine Nachbarschaft um P_{i_l} und P_{i_r} definiert. Dabei steht R für die Anzahl der zu berücksichtigenden Nachbarpixel, welche sich in x- bzw. y-Richtung um den Punkt $P_{i_{l/r}}$ herum befinden. Die so deklarierte Nachbarschaft enthält insgesamt $(2R+1)^2$ Bildpunkte (siehe Abbildung 2.13). Für die Ähnlichkeitsanalyse werden die Farb- bzw. RGB-Werte dieser Punkte in zwei Vektoren V_{i_l} und V_{i_r} der Dimension $(3 * (2R+1)^2)$ zusammengefasst. Der Faktor 3 ergibt sich durch die einzelnen Farbkomponenten Rot, Grün und Blau. Mithilfe der Normalisierten Kreuzkorrelation (NCC) kann der Ähnlichkeitsgrad zwischen den Vektoren V_{i_l} bzw. V_{i_r} für einen Punkt P_i folgendermaßen berechnet werden:

$$NCC(P_i) = \frac{V_{i_l} \cdot V_{i_r}}{\|V_{i_l}\| \, \|V_{i_r}\|} \ . \tag{2.42}$$

Da es sich hierbei um ein Skalarprodukt handelt, sind sich zwei Punkte P_{i_l} und P_{i_r} um so ähnlicher, desto höher der entsprechende $NCC(P_i)$-Wert ausfällt. Für die folgenden Verarbeitungsschritte wird der ermittelte Ähnlichkeitswert im dreidimensionalen Gitter an der Position des ursprünglich betrachteten Bildpunktes P_i in der Tiefenebene i hinterlegt.

Hinsichtlich der Implementierung können die drei Phasen des naiven Ansatzes folgendermaßen zusammengefasst werden: Zunächst wird für die Farbwertinterpolation im linken sowie rechten Eingabebild jeweils ein temporärer Vektor I_l bzw. I_r angelegt. Die Dimension der Hilfsvektoren entspricht dabei der Auflösung der einzelnen Gitterebenen (Höhe $\times$ Breite). Pro Tiefenebene i des 3D-Gitters wird jeder enthaltene Bildpunkt P_i (gegeben durch die x- und y-Koordinaten) in das Weltkoordinatensystem transferiert. Dazu wird der Abstand d_i zwischen dem Zentrum der virtuellen Kamera und der Tiefenebene i als z-Koordinate angenommen. P_i wird im Anschluss in das linke bzw. rechte Originalbild

projiziert. Für die resultierenden Projektionen P_{i_l} bzw. P_{i_r} werden die Farbwerte extrahiert und bzgl. der ursprünglichen x- und y-Koordinate in den Hilfsfeldern I_l bzw. I_r abgelegt. Sind alle notwendigen Farbwerte interpoliert worden, werden pro Punkt P_i die zugehörigen Vektoren V_{i_l} bzw. V_{i_r} angelegt. Diese werden mit den in I_l und I_r gesicherten Farbwerten von P_{i_l} bzw. P_{i_r}, inklusive derer aus ihrer Nachbarschaft, komponentenweise gefüllt und zur Berechnung des Ähnlichkeitsgrades $NCC(P_i)$ verwendet. Das Resultat wird in der Tiefenebene i an der Position P_i gesichert. Ein Überblick über die Korrespondenzsuche mithilfe des naiven Ansatzes ist in Algorithmus 1 zu finden. Für eine Parallelisierung sind die inneren for-Schleifen über die einzelnen Bildpunkte P_i einer Tiefenebene i ausschlaggebend.

```
foreach disparity layer i (with 0 ≤ i < N)
    foreach vertex P_i(x, y) in parallel do
        P_{i_world} = calculateWorldCoordinates(x, y, d_i)
        P_{i_l} = projectPoint(P_{i_world}, camera_left)
        P_{i_r} = projectPoint(P_{i_world}, camera_right)
        store(getInterpolateColor(P_{i_l}, I_l)
        store(getInterpolateColor(P_{i_r}, I_r)
    foreach vertex P_i(x, y) in parallel do
        V_{i_l} = getColorValuesOfNeighborhood(P_{i_l}, I_l)
        V_{i_r} = getColorValuesOfNeighborhood(P_{i_r}, I_r)
        normalize(V_{i_l})
        normalize(V_{i_r})
        storeInGrid(x, y, i, getDotProduct(V_{i_l}, V_{i_r}))
```

Algorithmus 1: Parallele Umsetzung des Naiven Ansatzes zur Korrespondenzsuche.

Nachdem der Basisalgorithmus zur Korrespondenzsuche erläutert wurde, sollen im Folgenden mögliche Erweiterungen bzw. Optimierungen zur Berechnung der Ähnlichkeitswerte erörtert werden. Da bei der Entwicklung des Verfahrens zur 3D-Rekonstruktion viel Wert auf eine hohe Parallelität gelegt wurde, ist die im Algorithmus 1 erkennbare Unterteilung der Ähnlichkeitsberechung in zwei for-Schleifen inakzeptabel. Gesucht war eine Möglichkeit diese insoweit zu kombinieren, dass alle drei Phasen der Korrespondenzsuche innerhalb eines Schleifenkonstruktes durchlaufen werden können. In diesem Sinne soll nun auf eine Vorverarbeitung der Bilddaten mithilfe einer Wavelet-Transformation (siehe Kapitel 2.1.5) eingegangen und deren Auswirkungen auf die Parallelisierung des Verfahrens diskutiert werden.

02.
Korrespondenzsuche **mittels Wavelet-Transformation**

Da während einer Wavelet-Transformation die gegebenen Bilddaten in mehrere Skalierungsstufen, inklusive der zugehörigen Hoch- bzw. Tiefpassdifferenzen zur nächsthöheren Auflösung, zerlegt werden, ist die Wahl der Waveletbasis für die Analyse der Koeffizienten (*Multiskalenanalyse* bzw. engl. *multi resolution analysis* - MRA) maßgeblich. Dabei sollten Eigenschaften wie Orthogonalität und Symmetrie sowie das Kompressionsverhalten einer Waveletfamilie berücksichtigt werden. Da es hier vornehmlich um die algorithmischen Veränderungen durch eine in das Verfahren integrierte komplexe Wavelet-Transformationen

(engl. *complex wavelet transformation* - CWT) gehen soll, wird an dieser Stelle auf eine Diskussion über die Vor- und Nachteile einzelner Wavelets verzichtet.

In Hinblick auf die Korrespondenzsuche müssen für die Projektionen P_{i_l} und P_{i_r} eines Bildpunktes P_i aus dem gegebenen reziproken Bildpaar Pixelinformationen extrahiert und über eine Ähnlichkeitsfunktion bewertet werden. Bei einer CWT werden während der Generierung eines Skalierungslevels j (mit $0 \leq j < M$) die Pixelinformation eines Bildpunktes mit denen der Waveletfunktion verrechnet, um die entsprechenden LL-, LH-, HL- und HH-Koeffizienten zu erhalten. Die LL-Koeffizienten werden für die Berechnung des nächsten Levels verwendet, wohingegen die anderen Werte die Differenzen zur nächst höheren Auflösung repräsentieren und nicht weiter betrachtet werden.

In diesen Berechnungszyklus kann die Korrespondenzsuche integriert werden. Angewendet auf das linke bzw. rechte Eingabebild können pro Skalierungsstufe j und Tiefenebene i die relevanten Pixelinformationen für die Projektionen P_{i_l} und P_{i_r} eines Bildpunktes P_i gesammelt und als Feature-Vektoren V_{ij_l} und V_{ij_r} einer Ähnlichkeitsbewertung unterzogen werden. Als zu extrahierende Daten seien hier die LH-, HL- und HH-Koeffizienten genannt, die hinsichtlich der Aufteilung der Farbwerte in ihre Rot-, Grün- und Blau-Anteile, die Dimension der Feature-Vektoren auf 9 festlegen. An dieser Stelle ist die Anzahl der Vektorkomponenten völlig ausreichend. Eine Nachbarschaft muss nicht explizit definiert werden, um den globalen Zusammenhang für einen Gitterpunkt P_i zu erfassen. Vielmehr ist die Nachbarschaft durch die Wahl der Waveletbasis indirekt gegeben und fließt bereits bei den Berechnungen der Waveletkoeffizienten mit ein. Mithilfe der Vektoren V_{ij_l} und V_{ij_r} kann im Anschluss der $NCC(P_i)$-Wert für den Bildpunkt P_i ermittelt und im Gitter abgespeichert werden. Da während der CWT mehrere Skalierungsstufen j generiert werden, können die für einen Punkt P_i bestimmten $NCC(P_i)$-Werte (einer pro Skalierungsstufe j) an der zugehörigen Gitterposition aufsummiert und über die Anzahl der erzeugten Bildskalierungen M gemittelt werden. Demzufolge kann die Formel 2.42 folglich modifiziert werden:

$$NCC(P_i) = \sum_{j=0}^{M-1} \left(\frac{V_{ij_l} \cdot V_{ij_r}}{M \, \|V_{ij_l}\| \, \|V_{ij_r}\|} \right) \tag{2.43}$$

Mit dieser Vorgehensweise können die beiden inneren *for*-Schleifen des Naiven Ansatzes aufgelöst bzw. zu einer zusammengefasst werden. Zudem kann durch den Verzicht auf die temporären Felder I_l und I_r Speicherplatz eingespart werden. Algorithmus 2 soll die angesprochenen Änderungen verdeutlichen.

Wurde zu jedem Bildpunkt P und pro annehmbaren Tiefenwert i der Ähnlichkeitsgrad zwischen den Projektionen P_{i_l} und P_{i_r} bestimmt, könnten nun im einfachsten Fall alle Ebenen des Gitters durchlaufen und der maximale $NCC(P_i)$-Wert für den Bildpunkt P gesucht werden. Ist für P die Gitterebene mit der optimalen Ähnlichkeit gefunden worden, kann der entsprechende Tiefenwert, als Grauwert interpretiert, in der Tiefenkarte (engl. *depth map*) abgespeichert werden. Da der höchste $NCC(P_i)$-Wert pro Bildpunkt P

```
foreach scale level j (with 0 ≤ j < M)
    foreach disparity layer i (with 0 ≤ i < N)
        foreach vertex P_i(x, y) in parallel do
            P_{i_world} = calculateWorldCoordinates(x, y, d_i)
            P_{i_l} = projectPoint(P_{i_world}, camera_left)
            P_{i_r} = projectPoint(P_{i_world}, camera_right)
            V_{ij_l} = getWaveletCoefficents(P_{i_l})
            V_{ij_r} = getWaveletCoefficents(P_{i_r})
            normalize(V_{ij_l})
            normalize(V_{ij_r})
            storeInGrid(x, y, i, (getDotProduct(V_{ij_l}, V_{ij_r})/M))
```

Algorithmus 2: Korrespondenzsuche unter Verwendung einer Wavelet-Transformation.

lediglich dem lokalen und nicht zwangsläufig dem globalen Optimum entsprechen muss, was zu Fehlern bzw. Artefakten im rekonstruierten dreidimensionalen Modell führen kann, sollte der vielversprechenste Tiefenkandidat zu P unter der Berücksichtigung der Nachbarschaftsverhältnisse gesucht werden. Diese Suche kann auf das Problem des *minimalen Schnitts* bzw. *maximalen Flusses* innerhalb eines Graphen zurückgeführt und mithilfe von Graph-Cut-Algorithmen (siehe Kapitel 2.1.7) gelöst werden. Bevor jedoch näher auf entsprechende Vorgänge eingegangen wird, sollen im Folgenden notwendige Vorberechnungen erläutert werden.

2.4.1.4 Berechnung der Konfidenzmatrix (Normalenabschätzung)

Mithilfe der Konfidenz können Aussagen über die Qualität der durchgeführten Tiefenabschätzung getroffen, bzw. die Suche nach dem optimalen Tiefenkandidaten zu einem Bildpunkt vollzogen werden. Um über ein entsprechendes Kriterium zu verfügen, muss zu jedem Punkt P der Tiefenkarte die Konfidenz ermittelt werden. Hierbei wird die Differenz zwischen den beiden Maximalwerten aus allen im Tiefengitter hinterlegten Ähnlichkeiten $val_i(P)$ mit $0 \leq i < N$, die P zugeordnet werden können, bestimmt und letztlich für die Gewichtung des *Best Match* verwendet (siehe Algorithmus 3).

```
float m_1, m_2, tmp
foreach vertex P(x, y)
    foreach disparity layer i (with 0 ≤ i < N)
        tmp = getGridValue(x, y, i)
        if (m_1 < tmp)
            m_2 = m_1
            m_1 = tmp
    storeConfidence(x, y, m_1 − m_2)
```

Algorithmus 3: Die Berechnung der Konfidenzmatrix.

Unterliegen die für P berechneten Ähnlichkeitswerte einem starken Streuungsgrad, fallen entsprechende Differenzwerte bzw. Konfidenzen hoch aus, was auf eine akzeptable

Tiefenschätzung hindeutet. Sind hingegen geringe Abweichungen zu verzeichnen, liegen die ermittelten Ähnlichkeiten nah beieinander, was eine genauere Differenzierung unmöglich macht. Der höchste zwischen zwei Projektionspunkten bestimmte Ähnlichkeitsgrad verliert in diesem Fall an Aussagekraft.

Für einige Teile der folgenden Verfahrensschritte ist es wichtig, pro Bildpunkt und angenommener Tiefe bzw. Disparität i eine Normalenabschätzung durchzuführen. Mithilfe der Normalen kann beispielsweise die Bestimmung der Ähnlichkeiten optimiert werden (siehe Abschnitt 2.4.1.5). Da für die Rekonstruktion dreidimensionaler Strukturen reziproke Bildpaare verwendet werden, kann die Formel 2.18 (siehe Kapitel 2.1.3) zur Normalenabschätzung herangezogen werden, indem sie nach $\mathbf{n}$ aufgelöst wird.

Sei O_l bzw. O_r der zur linken bzw. rechten Bildkamera gehörende Koordinatenursprung und P_w ein Punkt auf der Objektoberfläche, inklusive der korrespondierenden Normalen $\mathbf{n}$. Die Richtung zwischen P_w und O_l bzw. O_r wird durch die beiden Vektoren $\mathbf{v}_l = \frac{1}{|O_l - P_w|}(O_l - P_w)$ und $\mathbf{v}_r = \frac{1}{|O_r - P_w|}(O_r - P_w)$ repräsentiert. Mithilfe dieser Komponenten kann die Beleuchtungsintensität für die Projektionen von P_w folgendermaßen aus den Eingabebildern extrahiert werden [ZHK*03]:

$$e_l = f_r(\mathbf{v}_r, \mathbf{v}_l)\frac{\mathbf{n} \cdot \mathbf{v}_r}{|O_r - P_w|^2} \tag{2.44}$$

wobei $\mathbf{n} \cdot \mathbf{v}_r$ dem Cosinus des Winkels zwischen dem Richtungsvektor zur Lichtquelle und der Normalen, bzw. f_r der bidirektionalen Reflektanzverteilungsfunktion (engl. *bidirectional reflectance distribution function* - BRDF) entspricht.

Da laut der Reziprozität zwei korrespondierende Bildpunkte lediglich von der Beschaffenheit der Objektoberfläche abhängig und gegenüber vorhandenen Beleuchtungseffekten unempfindlich sind, kann diesbezüglich die Symmetrieeigenschaft der BRDF angenommen werden. In einem solchen Fall kann die BRDF vernachlässigt werden, worauf letzlich die Herleitung der Formel 2.18 zurückgeführt werden kann [MKZB01]:

$$\left(e_l \frac{\mathbf{v}_l}{|O_l - P_w|^2} - e_r \frac{\mathbf{v}_r}{|O_r - P_w|^2}\right) \cdot \mathbf{n} = 0.$$

Um den durch die Normalenabschätzung implizierten Fehler zu reduzieren, sollte die Differenz zwischen den Beleuchtungstermen, die sich im Zusammenhang mit Formel 2.44 und der ermittelten Normalen $\mathbf{n}$ aus der BRDF des linken bzw. rechten Kamerabildes ergibt, minimiert werden.

2.4.1.5 Die Suche nach dem *Best Match*

Bevor auf die Suche nach dem *Best Match* genauer eingegangen wird, soll eine Optimierungsstrategie diskutiert werden, mit welcher Mehrdeutigkeiten reduziert bzw. mögliche Fehleinschätzungen bei der Wahl des geeignetsten, zu einem Bildpunkt P gehörenden

Tiefenkandidat vermieden werden können. Da bei der Ähnlichkeitsanalyse auf die *Normalisierte Kreuzkorrelation* (NCC) zurückgegriffen und bei den entsprechenden Berechnungen die Nachbarschaft $N(P)$ eines Pixels P mit einbezogen wird, kann es trotz einer größeren Anzahl von berücksichtigten Pixelinformationen (Farbintensität, Waveletkoeffizienten usw.) aus $N(P)$ zu Mehrdeutigkeiten bei den ermittelten NCC-Werten kommen. Die Wahrscheinlichkeit, dass sich für verschiedene Tiefenwerte bzw. Disparitäten ähnliche NCC-Werte ergeben, sinkt mit der steigenden Anzahl betrachteter Nachbarschaftselemente. Dieser Umstand kann jedoch nicht komplett ausgeschlossen werden, wodurch Fehlinterpretationen oder Artefakte im resultierenden 3D-Modell möglich werden.

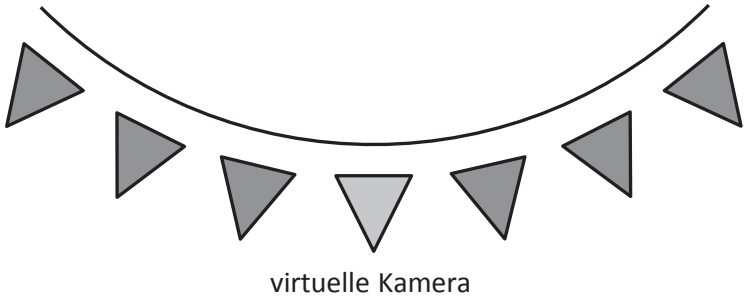


Abbildung 2.14: Schematische Darstellung zur Positionierung mehrerer Kameras.

01. Verbesserung **mittels mehrerer reziproker Bildpaare**

Eine einfache Möglichkeit das Aufkommen von Mehrdeutigkeiten zu verringern, stellt die Verwendung zusätzlicher, reziproker Bildpaare dar, die aus unterschiedlichen Kamerapositionen heraus aufgenommen wurden. Dabei sollte darauf geachtet werden, dass die einzelnen Bilder in einem ähnlichen Abstand zum Objekt hin aufgenommen wurden und keinerlei Überdeckungen oder andere als die zu rekonstruierenden Elemente enthalten.Ersteres kann mithilfe des Winkels zwischen dem Richtungsvektor der virtuellen Kamera und denen der Bildkameras reguliert werden. Die Abbildung 2.14 soll diesen Sachverhalt schematisch veranschaulichen.

Die Auswirkungen auf den Algorithmus beschränken sich darauf, dass nach der Definition des Tiefengitters die verschiedenen Bildpaare mittels einer Schleife eingelesen und verarbeitet werden. Die dabei ermittelten Ähnlichkeitswerte werden pro Bildpunkt für jeden festgelegten Tiefenwert aufsummiert und an entsprechender Stelle im Gitter hinterlegt. Durch diese Herangehensweise wird kein zusätzlicher Speicherplatz auf der GPU benötigt. Es kommt lediglich zu einem erhöhten Datentransfer, da die zusätzlichen Bilddaten inklusive der korrespondierenden Kameraeigenschaften geladen werden müssen.

02. Verbesserung **mithilfe einer Wavelet-Transformation**

Um Ungenauigkeiten während der Berechnung der *Normalisierten Kreuzkorrelation* (siehe Kapitel 2.1.6.3) auszugleichen oder Folgefehlern vorzubeugen, könnte ergänzend auf jede der angelegten Tiefenebenen eine Wavelet-Transformation angewendet werden. Dies würde eine Stabilisierung der Ähnlichkeitswerte bewirken, da aneinander angrenzende

NCC-Werte zueinander in Relation gesetzt werden. Der sich daraus ergebende Vorteil wird jedoch durch den erhöhten Rechenaufwand geschmälert. Im entwickelten Verfahren wurde deswegen auf diese Möglichkeit verzichtet.

Wie bereits erwähnt, könnte nach der Korrespondenzsuche zur Generierung einer Tiefenkarte pro Bildpunkt P der Tiefenwert aus dem 3D-Gitter gewählt werden, für welchen der höchste *NCC*-Wert ermittelt wurde. Als Grauwerte interpretiert entsteht so ein Graustufenbild, welches zur Konstruktion eines dreidimensionalen Polygonnetzes verwendet werden kann. Da sich hierbei die Rekonstruktion eines Objektes ausschließlich auf die Betrachtung der lokalen Optima beschränkt, ohne das Verhältnis zur Nachbarschaft miteinzubeziehen, können benachbarte Pixel, die sich im Originalobjekt innerhalb einer Ebene befinden, aufgrund der lokal analysierten *NCC*-Werte unterschiedliche Tiefenwerte annehmen und dadurch kegelartige Auswüchse bzw. scharfe Kanten im generierten 3D-Modell hervorrufen. Um derartige Artefakte zu vermeiden, müssen die ermittelten Ähnlichkeitswerte in einem globalen Zusammenhang gesehen und ausgewertet werden. Hierfür kann die Suche nach dem *Best Match* auf das Problem des *maximalen Flusses* bzw. *minimalen Schnitts* innerhalb eines Graphen (siehe Kapitel 2.1.7) zurückgeführt werden und das definierte Tiefengitter als Eingabe für einen Graph-Cut-Algorithmus fungieren.

01. *Best-Match*-Suche **über Minimierung der $L1$-Norm**

Da die Parallelisierung eines Graph-Cut-Algorithmus eine komplexe Aufgabe darstellt, wurde anfänglich eine Strategie verfolgt, welche auf einen Artikel von Arvind Bhusnurmath und Camillo J. Taylor [BT08] zurückgeführt werden kann. Der darin beschriebene Ansatz besagt, dass die Berechnung des *minimalen Schnitts* auf die Minimierung der $L1$-Norm einer Matrix reduziert werden kann, sofern der jeweilige Graph in eine entsprechende Matrixform überführt wird bzw. die einzelnen Matrixeinträge mit den Kantenkapazitäten eines Graphen korrespondieren. Der Beweis, dass der *minimale Schnitt* eines Graphen ein Äquivalent zur minimalen $L1$-Norm der zugehörigen Graphenmatrix darstellt, wurde von Ali K. Sinop und Leo Grady [SG07] geführt.

Im Originalverfahren von Bhusnurmath und Taylor wird eine Matrix von der Größe $\#Kanten * (\#Knoten + 2)$ generiert, wobei jede Zeile eine Kante und jede Spalte einen bestimmten Knoten repräsentiert. Zusätzlich werden zwei vituelle Knoten S (Quelle) und T (Senke) berücksichtigt. Da durch jede Kante zwei Knoten miteinander verbunden werden, enthält jede Zeile lediglich zwei von Null verschiedene Kapazitätswerte. Die Knoten, die mit S bzw. T verknüpft werden, verfügen über eine maximale Kapazität. Im Gegensatz zu dieser speicherintensiven Variante wurde im entwickelten Verfahren eine Matrizenstruktur gewählt, in welcher pro Zeile eine Kante mithilfe zweier Knotenindizes und der zugehörigen Kapazität definiert wird. Mit dieser Vorgehensweise kann die Größe der Graphenmatrix auf $\#Kanten * 3$ reduziert werden.

Wurde die Korrespondenzsuche erfolgreich abgeschlossen, lässt sich die Anzahl der Knoten und Kanten aus den Dimensionen des Tiefengitters ableiten:

$$\begin{aligned} \#Knoten &= Breite * Höhe * Tiefe && (2.45) \\ \#Kanten &= (Breite - 1) * Höhe * Tiefe \\ &+ Breite * (Höhe - 1) * Tiefe && (2.46) \\ &+ Breite * Höhe * (Tiefe - 1). \end{aligned}$$

Für die Berechnung der Kantenkapaziäten $cap(P_{i_1}, P_{i_2})$ werden die zu den beiden Gitterpunkten P_{i_1} bzw. P_{i_2} in der Tiefenebene i befindlichen Ähnlichkeitswerte $NCC(P_{i_1})$ und $NCC(P_{i_2})$ ausgelesen, mit einer benutzerdefinierten Konstante ω gewichtet und über die Formel

$$cap(P_{i_1}, P_{i_2}) = e^{-\omega * NCC(P_{i_1})} + e^{-\omega * NCC(P_{i_2})} \tag{2.47}$$

miteinander verrechnet. Um den Bezug zwischen den Knoten P_i und P_j zweier unterschiedlicher Tiefenebenen i bzw. j darzustellen, wurde folgende Formel verwendet:

$$cap(P_i, P_j) = s * (e^{-\omega * NCC(P_i)} + e^{-\omega * NCC(P_j)}) \ . \tag{2.48}$$

Dabei beschreibt s die Glattheit (engl. *smoothness*) der resultierenden Tiefenkarte.

Nachdem die Graphen- bzw. Kantenmatrix aufgestellt wurde, kann diese dazu verwendet werden, das folgende Minimierungsproblem zu lösen:

$$\arg\min_x \|b - Ax\|_1 = \arg\min_x \sum_{i=1}^{n} |b_i - A_i x|^1. \tag{2.49}$$

Mithilfe der Methode zur iterativen Bestimmung der kleinsten, gewichteten Quadrate (engl. *iteratively reweighted least squares* - IRLS) kann der Lösungsvektor $x = (x_1, x_2, \cdots, x_m)$ über die Kantenmatrix A und $b = Ax$ berechnet, bzw. iterativ angenähert werden:

$$x^{(t+1)} = \arg\min_x \sum_{i=1}^{n} \omega_i^{(t)} |b_i - A_i x|^2, \tag{2.50}$$

wobei $\omega_i^{(t)}$ eine Diagonalmatrix zur Gewichtung repräsentiert und im Initialisierungsschritt der Einheitsmatrix entspricht. Die Einträge der Gewichtungsmatrix werden während der Iterationen aktualisiert und lassen sich wie folgt ermitteln:

$$\omega_i^{(t)} = \frac{1}{|b_i - A_i x|} \ . \tag{2.51}$$

Da für die Suche nach dem *Best Match* bzw. optimalen Tiefenwert eines Bildpunktes keine exakte Lösung des genannten Minimierungsproblems vorliegen muss, kann der Aufwand für die zum Lösen des pro Iterationschritt der IRLS-Methode aufgestellten

Gleichungssystems $\omega Ax = \omega b$ notwendigen und rechenintensiven Kalkulationen mithilfe des Verfahrens der konjugierten Gradienten (engl. *conjugate gradients* - CG) gesenkt werden. Der Vorteil des CG-Verfahrens liegt darin, dass die Minimierung des Funktionals

$$f(x) = \frac{1}{2}x^T Ax - b^T x \tag{2.52}$$

äquivalent zur Lösung des Gleichungssystems $Ax = b$ ist. Wird deren erste Ableitung $f'(x) = Ax - b$ betrachtet, kann ein minimaler Vektor x mit der Aussage $f'(x) \approx 0$ gleichgesetzt werden. Dies bedeutet wiederum, dass eine Lösung für das obige Gleichungssystem gefunden wurde. Ein weiterer Vorteil der CG-Methode ist die einfache Parallelisierbarkeit der voneinander unabhänigigen Brechnungsschritte.

In der Summe ergeben das IRLS-Prinzip und der CG-Ansatz ein hochparalleles Verfahren, mit welchem die $L1$-Norm der Kantenmatrix A minimiert und der Graph-Cut letztendlich ersetzt werden kann. Die einzelnen Einträge im Lösungsvektor x stehen dabei für die verschiedenen Knoten des Graphen. Die Koeffizienten, die gegen 1 konvergieren, repräsentieren eine Verbindung zwischen dem zugehörigen Knoten und der Quelle S. Konvergiert eine Komponente von x gegen 0 sind entsprechende Knoten mit der Senke T verbunden. Aufgrund dieses Separationsverhaltens können Rückschlüsse auf den *minimalen Schnitt* bzw. *maximalen Fluss* eines Graphen gezogen werden.

02.
Best-Match-Suche **mittels parallelem Push-Relabel**

Um die Vor- und Nachteile gegenüber einem Graph-Cut-Verfahren abwägen zu können, wurde zusätzlich der von Andrew V. Goldberg und Robert E. Tarjan entwickelte *Push-Relabel*-Algorithmus [GT88] implementiert (siehe Kapitel 2.1.7.2). Die Wahl fiel auf die *Push-Relabel*-Methode, da diese zu den effizientesten und stabilsten Algorithmen im Bereich des Graph-Cuts zählt und bereits viele Überlegungen hinsichtlich der Parallelisierbarkeit dieses Verfahrens existieren [GT88, HH10, VN08].

Für die Parallelisierung des *Push-Relabel* Algorithmus sollten alle im Graphen enthaltenen Knoten parallel verarbeitet werden können. Dem besagten Verfahren liegen zwei voneinander unabhängige Operationen zugrunde: die *Push*-Operation, bei welcher ein benachbarter Knoten den verfügbaren Überfluss erhält und die *Relabel*-Operation, bei welcher die Höhe des Knotens angepasst wird, um den eintrefenden Fluss weiterleiten zu können. Erstere kann nur dann ausgeführt werden, wenn ein positiver Überfluss vorhanden und die Höhe des Knotens mindestens um 1 höher ist als die eines Nachbarknotens. Da sich laut He und Hong [HH10] keine nennenswerten Vorteile im Bezug auf die Parallelisierung des Intialsierungsschrittes (siehe Kapitel 2.1.7.2) ergeben, beschränkt sich die folgende Diskussion auf die Parallelisierungsmöglichkeiten der Grundoperationen.

Bei der parallelen Ausführung einer *Push*-Operation muss beachtet werden, dass während der Übertragung von vorhandenem Fluss auf einen Nachbarknoten, ein weiterer Knoten auf eben denselben Nachbarn seinen Überschuss schicken kann. Da ein Knoten nur eine bestimmte Anzahl von Fluss aufnehmen kann, muss vor jeder Transaktion die Restkapazität

erfragt werden. Wird eine derartige Anfrage gestellt bevor der Fluss eines anderen Knotens eingetroffen ist, werden falsche Angaben zwischen einzelnen Knoten versendet. Um dieses Problem zu lösen, können zwei Strategien verfolgt werden. Zum einen könnte die *Push*-Operation in zwei Phasen, die *Bring*- und *Holphase*, unterteilt oder zum anderen exklusive Lese- und Schreibrechte vergeben werden. Für Ersteres muss zwischen zwei Knoten eine Art Reservoir oder Container angelegt werden, in welchem der Überfluss eines Knotens im Zuge der *Bringphase* geschoben und von seinem Nachbarknoten während der nachfolgenden *Holphase* ausgelesen werden kann. In der Holphase kann jeder Knoten soviel Fluss aus den umliegenden Containern aufnehmen, wie es laut dessen Restkapazitäten möglich ist. Durch diese Vorgehensweise steigt jedoch die Anzahl der Zugriffe um das Doppelte, da jeder Knoten prüfen muss, ob sein Fassungsvermögen für den gelieferten Fluss ausreicht oder ob jedes angrenzende Reservoir überhaupt Fluss enthältet. Für die Parallelisierung ergibt sich somit keine nennenswerte Zeitersparnis [HH10]. Bei der Vergabe von exklusiven Lese- und Schreibrechten darf ein Knoten erst eine Anfrage an seinen Nachbarn schicken, wenn von anderer Stelle an diesen kein Fluss mehr übertragen wird bzw. wenn die Aktualisierung der Restkapazität abgeschlossen ist.

Die Parallelisierung der *Relabel*-Operation gestaltet sich hingegen komfortabler, da hier die Höhen der einzelnen Knoten lediglich gegenüber ihren Nachbarn angepasst werden müssen. Werden zunächst die Höhen aller Nachbarknoten, zu denen eine Restkante führt, erfragt und deren Minimum gewählt, kann dieses im Anschluss um 1 inkrementiert und als neue Höhe angenommen werden. Durch diese Herangehensweise werden mögliche Konflikte vermieden. Es kann jedoch passieren, dass der Fluss zwischen zwei Knoten oszilliert und die jeweiligen Höhen bis zum Maximum angehoben werden, weshalb dieses Vorgehen im *worst case* ineffizient ist.

Aus den geschilderten Parallelisierungsmöglichkeiten der *Push*- bzw. *Relabel*-Operationen ergibt sich der folgende grundlegende Ablauf: Nach dem Intialisierungschritt wird für jeden Knoten überprüft, ob dieser einen Fluss weiterleiten kann. Ist dies der Fall, wird eine parallele *Push*-Operation durchgeführt, andernfalls ein paralleles *Relabeling* vollzogen. Dieser Vorgang wird solange wiederholt, bis die Abbruchbedingung greift, also kein Fluss mehr zwischen Knoten verschickt, die Restkapazitäten der Knoten ausgeschöpft oder die maximale Anzahl von Iteration erreicht wurde. Letzteres lässt sich durch das maximale Aufkommen von $Nachbarknoten * 2$ festlegen, da jeder Knoten seinen gesamten Überfluss an seine Nachbarn abgeben und dazu ggf. seine Höhe anpassen muss. Da es sich hierbei jedoch nur um eine obere Schranke handelt, sollte nach jeder Iteration getestet werden, ob der in der Quelle S und der Senke T eingehende Überfluss in der Summe dem anfänglich von der Quelle S ausgesendeten Gesamtfluss entspricht. Ist diese Bedingung erfüllt, kann der Algorithmus terminieren und der in der Senke T befindliche Überfluss entspricht dem *maximalen Fluss* des Eingabegraphen. Der zugehörige *minimale Schnitt* enthält die optimalen Tiefenkandidaten eines jeden Bildpunktes. Diese können nun ausgelesen und in Form einer Tiefenkarte, d.h. als Graustufenbild, abgespeichert werden.

Um die Effizienz der zu tätigenden Abfragen zu verbessern, wurde eine Datenstruktur eingeführt, welche die Relationen zwischen zwei Knoten repräsentiert. Jeder Relation werden die entsprechenden Indizes der Knoten sowie die Kapazitäten der beiden möglichen Transferrichtungen zugeordnet. Für jeden Knoten wird ein Feld für die Höhe und eines für den Überfluss angelegt. Die üblicherweise verwendete matrizenartige Struktur enthält hingegen aufgrund der Betrachtung von nicht vorhandenen Knotenverbindungen viele Nulleinträge, was neben der unnötigen Speicherauslastung zu erheblichen Performanzeinbußen führt.

Aufgrund der Tatsache, dass mit steigender Auflösung der Originalbilder die Dimensionen des Tiefengitters zunehmen und der Rechenaufwand, der für den Graph-Cut benötigt wird, von der Komplexität des Gitters abhängig ist, sind entsprechende Berechnungen für die Programmperformanz entscheidend. Auch wenn sämtliche Optimierungsmöglichkeiten für den Graph-Cut in Betracht gezogen werden, bleibt die besagte Abhängigkeit bestehen. Zudem lässt sich durch entsprechende Modifikationen lediglich eine geringe Zeitersparnis erzielen. Um diesen Aufwand sowie den damit einhergehenden Speicherbedarf zu reduzieren, wurde im Rahmen dieser Arbeit eine Technik entwickelt, welche auf dem Prinzip der *Belief Propagation* beruht.

Ganz allgemein können mithilfe der *Belief Propagation* Inferenzen innerhalb eines Graphenmodells bestimmt werden. Dabei wird zu jedem Knoten, unter Berücksichtigung seiner Nachbarschaft, dessen Marginalverteilung bzw. die zugehörige relative Häufigkeit eines in der näheren Umgebung aufkommenden Merkmals errechnet. Während der Berechnungen werden die Initialwerte der einzelnen Knotenelemente über einen Nachrichtenaustausch innerhalb des kompletten Graphen iterativ solange modifiziert, bis diese konvergieren.

Im Bezug auf die Rekonstruktion dreidimensionaler Objekte und der damit im Zusammenhang stehenden Bestimmung von Tiefenwerten kann das Schema der *Belief Propagation* mit der Korrespondenzsuche kombiniert werden. In einem solchen Fall kann pro Bildpunkt ein Tiefenwert angenommen und in Anlehnung an dessen Nachbarschaft relativiert werden. Innerhalb des definierten Tiefengitters können somit pro Tiefenebene die enthaltenen Ähnlichkeitswerte in einen konvergenten Zustand überführt werden. Aufgrund der durch die *Belief Propagation* hervorgerufenen Regularisierung ist es letztendlich möglich, zu jeden Bildpunkt direkt den optimalen Tiefenkandidat aus dem 3D-Gitter abzulesen, ohne dafür einen zusätzlichen Graph-Cut ausführen zu müssen.

Problematisch ist an dieser Stelle der Initialisierungsschritt. Einerseits sollte ein Initialwert die bestehenden Zusammenhänge möglichst genau widerspiegeln, um Berechnungsfehler vermeiden zu können und um diesen effizient zum Optimum hin konvergieren zu lassen. Auf der anderen Seite sollte die Gesamtperformanz nicht vernachlässigt werden. So besteht die Möglichkeit, die Eingabebilder in Bezug auf deren Auflösung auf ein Minimum zu reduzieren. Auf der niedrigsten Auflösungsstufe angekommen, kann mithilfe eines der zuvor geschilderten Ansätze zur Korrespondenzsuche für jeden Bildpunkt P_i der Initialwert

$NCC_i(P_i)$ ermittelt und pro veranschlagter Disparität i im Tiefengitter hinterlegt werden.

Für das entwickelte Verfahren wurde jedoch in Hinblick auf das Konvergenzproblem ein Abstandsmaß zur Fehlerminimierung definiert. In diesem Sinne werden zunächst zu jedem Bildpunkt P_i einer Tiefenebene i die Farbinformationen aus den Eingabebildern extrahiert und zur Abschätzung der Punktnormalen $\mathbf{n_{P_i}}$ verwendet (siehe Abschnitt 2.4.1.4). Mithilfe der Normalen $\mathbf{n_{P_i}}$ wird anschließend der Beleuchtungsfehler err_{P_i} berechnet, indem die Diskrepanz zu der den Eingabebildern zugehörigen *bidirektionalen Reflektanzverteilungsfunktion* (BRDF) an den jeweiligen Bildpositionen ermittelt wird. Dieser Fehler wird als Initialwert für die weiteren Schritte angenommen.

Nachdem die Ähnlichkeiten hinsichtlich der aktuell betrachteten Tiefe i analysiert wurden, kann der Wert von err_{P_i} in Abhängigkeit zur Nachbarschaft iterativ nach dem Prinzip der *Belief Propagation* in einen konvergenten Zustand (alle err_{P_i}-Werte konvergierten oder eine deklarierte maximale Anzahl von Iterationen wurde überschritten) transferiert werden. Wurde dies für jede der im 3D-Gitter befindlichen Tiefenebenen vollzogen, kann die Bildauflösung verdoppelt, unter Berücksichtigung der bereits vorhandenen Daten ein Initialwert für die neu hinzugekommenen Bildpunkte vermerkt und die Regulierung der einzelnen Werte wiederholt werden. Dieser Prozess wird solange durchlaufen bis ein konsistenter Zustand auf der höchsten Auflösungsstufe und damit für die ursprünglichen Bilddimensionen erreicht wurde. Abschließend kann für jeden Bildpunkt das Optimum aus dem Gitter gefiltert werden.

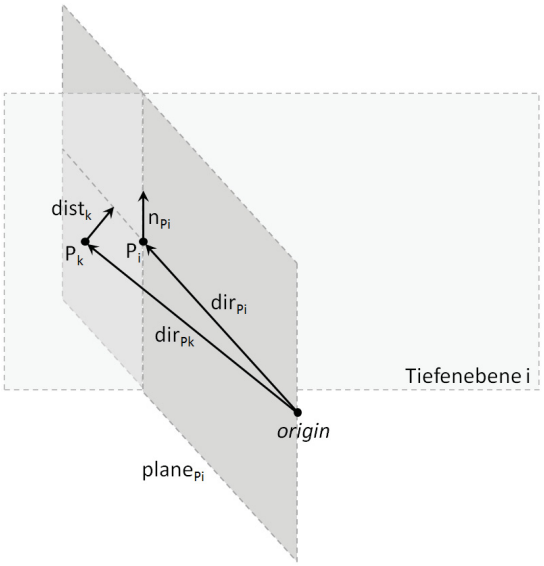


Abbildung 2.15: Schematische Darstellung zur Berechnung des Fehlerwertes err_{P_i}.

Im entwickelten Verfahren wird während der Propagationsphase folgendes Prozedere verfolgt (vgl. Abbildung 2.15): Um Konvergenzaussagen treffen zu können, müssen die aktuell berechneten Werte im Verhältnis zu denen der vorherigen Iteration betrachtet werden. Aus diesem Grund wird am Ende einer Iteration zu jedem Bildpunkt P der momentan beste Tiefenkandidat i aus dem Tiefengitter ausgelesen und zwischengespeichert (im Folgenden als $prevDepth_P$ bezeichnet). Im ersten Durchlauf wird hierfür lediglich der Ähnlichkeitsgrad bzgl. der Reziprozität bestimmt. Die Tiefe i, für welche der niedrigste Beleuchtungsfehler err_{P_i} eines Punktes P ermittelt wurde, wird als $prevDepth_P$ vermerkt. Ab der zweiten Iteration wird nach der Normalenabschätzung und im Bezug auf die aktuell veranschlagte Disparität i, zunächst der Richtungsvektor dir_{P_i} zwischen dem optischen Zentrum der virtuellen Kamera *origin* und dem Punkt P_i bestimmt, welcher im Anschluss dazu genutzt wird, um mithilfe der Punktnormalen n_{P_i} eine virtuelle Ebene $plane_{P_i}$ aufzuspannen. Sind diese Vorberechnungen abgeschlossen, werden die einzelnen Richtungsvektoren dir_{P_k}, der zu den direkt an P_i angrenzenden

und in der Tiefenebene i befindlichen Pixeln P_k (mit $0 \leq k < 8$), dazu verwendet, deren Abweichung $dist_k$ zur Ebene $plane_{P_i}$ zu erfassen, bevor diese anhand der zu jedem P_k gesicherten Werte $prevDepth_{P_k}$ in ein Verhältnis gesetzt und aufsummiert werden. Die resultierende Summe sum_{P_i} wird in Hinblick des aktuellen Auflösungsgrades gewichtet und auf den im Gitter an der Position P_i hinterlegten Fehlerwert err_{P_i} addiert:

$$sum_{P_i} = \sum_{k=0}^{7} max(prevDepth_{P_k}/dist_k, dist_k/prevDepth_{P_k}) - 1, \; mit \tag{2.53}$$

$$dist_k = ((dir_{P_i} - origin) \cdot n_{P_i})/(dir_{P_k} \cdot n_{P_i}). \tag{2.54}$$

Bei der Aktualisierung der Auflösung wird für die neu hinzugekommenen Bildpunkte P ein aus der direkten Nachbarschaft gewicheter Tiefenwert als $prevDepth_P$ festgesetzt.

```
foreach scale level j (with 0 ≤ j < M)
    loop until convergence
        foreach disparity layer i (with 0 ≤ i < N)
            foreach vertex P_i(x, y) in parallel do
                P_{i_world} = calculateWorldCoordinates(x, y, d_i)
                P_{i_l} = projectPoint(P_{i_world}, camera_left)
                P_{i_r} = projectPoint(P_{i_world}, camera_right)
                V_{ij_l} = getFeatureVector(P_{i_l})
                V_{ij_r} = getFeatureVector(P_{i_r})
                normalize(V_{ij_l})
                normalize(V_{ij_r})
                n_{P_i} = approximateNormal(P_{i_world}, V_{ij_l}, V_{ij_r})
                err_{P_i} = getIlluminationError(P_{i_world}, V_{ij_l}, V_{ij_r}, n_{P_i})
                if (j ≠ 0)
                    plane_{P_i} = calculateVirtualPlane(P_{i_world}, n_{P_i})
                    sum_{P_i} = getVariation(plane_{P_i}, prevDepth_P)
                    err_{P_i} = err_{P_i} + sum_{P_i}
                storeInGrid(x, y, i, err_{P_i})
        foreach vertex P(x, y) in parallel do
            prevDepth_P = getOptimalDepthCandidate()
```

Algorithmus 4: Die parallele Berechnung der Tiefenwerte inklusive der Suche nach dem *Best Match*, basierend auf dem Grundprinzip der *Belief Propagation*.

Wird dieser Ansatz zur Tiefensuche gewählt, ergibt sich ein weiterer Vorteil dadurch, dass für jeden Bildpunkt P ein Intervall abgeschätzt werden kann, in welchem sich der optimale Tiefenwert befinden muss. Diese Abschätzung geschieht während der Normalenberechnung, bei welcher der Beleuchtungsfehler ermittelt wird. Da der entsprechende Beleuchtungsterm im Intervall $[0, 1]$ liegen muss, können alle Tiefenwerte i, für welche die Punktnormale $\mathbf{n_{P_i}}$ des Bildpunktes P bestimmt wurde, vernächlässigt werden, wenn der mittels $\mathbf{n_{P_i}}$ errechnete Term außerhalb des angegebenen Wertebereiches liegt. Dadurch können die Tiefenberechnungen auf das Notwendigste beschränkt werden, was zu einer Steigerung der Performanz führt. In Algorithmus 4 wird das beschriebene Vorgehen zusammengefasst.

2.4.1.6 Verwendete Datenstruktur

Im Folgenden soll ein Überblick über die benötigte Datenstruktur zur parallelen Generierung der Tiefenkarten gegeben werden. Dabei beschränken sich die Angaben auf die Verfahrensvariante, bei welcher die Korrespondenzanalyse und die Suche nach dem *Best Match* nach dem Grundprinzip der *Belief Propagation* kombiniert werden können (siehe Algorithmus 4).

Bilddaten: Während der Initialisierungsphase werden die Daten eines reziproken Bildpaares geladen und anschließend auf die Grafikkarte transferiert. Um die Ähnlichkeitsbewertung zu optimieren, werden für die Berechnungen mehrere Bildpaare berücksichtigt. An dieser Stelle sei darauf hingewiesen, dass abweichend vom Vermerk zum Thema *Verbesserung mittels mehrerer reziproker Bildpaare* im Abschnitt 2.4.1.5, nach welchem jedes Bildpaar für sich verarbeitet werden kann und somit kein zusätzlicher Speicher beansprucht wird, im entwickelten Algorithmus für einen effizienten Verfahrensablauf sämtliche Paare auf einmal geladen werden. Der Grund hierfür liegt im Verfahren selbst und wird im Zusammenhang mit dem für das Tiefengitter relevanten Speicherbedarf diskutiert. Neben den Originalaufnahmen werden zwei Bilder, welche für die Entzerrung der Eingabebilder verwendet werden, sowie zwei Hilfsfelder für die Projektion aller Bildpunkte aus der virtuellen Kamera in das linke bzw. rechte entzerrte Bild und die damit verknüpfte Farbwertinterpolation angelegt.

Kameraeigenschaften: Für die korrekte Entzerrung der Bilder, für die Projektion der einzelnen Bildpunkte sowie für die Ähnlichkeitsanalyse hinsichtlich der Reziproziät sind die zu den geladenen Bildpaaren gehörenden intrinsischen und extrinsischen Koeffizienten unentbehrlich. Diese werden zusammen mit den Originalaufnahmen geladen und auf die Grafikkarte übertragen. Entsprechend der Anzahl an betrachteten reziproken Bildpaaren muss Speicher für die jeweiligen Rotationsmatrizen und Translationsvektoren sowie für die zu den drei Kameras gehörenden intrinsischen Koeffizienten veranschlagt werden. Da für die virtuelle Kamera das ideale Lochkameramodell (siehe Abschnitt 2.1.1.1) angenommen werden kann, entfällt hier der Speicherbedarf für die Verzerrungskoeffizienten.

Tiefengitter: Laut Algorithmus 4 wird für jeden angenommenen Tiefenwert i die Ähnlichkeit für alle Bildpunkte P berechnet und nach jeder Iteration der momentan beste Tiefenkandidat für P bestimmt. Dieser letzte Schritt kann während der Betrachtung der einzelnen Tiefenebenen erfolgen, indem zu jedem P der minimale Fehlerwert der zuvor analysierten Tiefen in Verbindung mit der zugehörigen Disparität in der Datenstruktur vermerkt und mit dem der aktuellen Ebene verglichen wird. Somit muss keine dreidimensionale Gitterstruktur aufgebaut und über die gesamte Programmlaufzeit verwaltet werden. Es müssen lediglich drei Hilfsfelder in Abhängigkeit zur Auflösung der Bilddaten angelegt werden, wobei eines der Felder

für die Korrespondenzsuche innerhalb der aktuell betrachteten Tiefenebene i und die beiden anderen zum Speichern des für einen Bildpunkt P, in einer vorherigen Iteration ermittelten minimalen Fehlers und der zugehörigen Disparität benötigt werden. Wird kein dreidimensionales Tiefengitter angelegt, müssen wiederum in Hinblick auf die Reziprozitätsanalyse, bei welcher die Farbinformationen aus den Originalbildern extrahiert werden, sämtliche Bildpaare verfügbar sein, um den zu ermittelnden Fehler für einen Bildpunkt P aufsummieren zu können. Da davon auszugehen ist, dass die Anzahl der eingelesenen Bildpaare weit unter der Anzahl der betrachteten Disparitäten i liegt, muss für die reziproken Bildpaare weitaus weniger Speicher alloziert werden. Durch das entwickelte Verfahren kann also nicht nur ein Performanzgewinn durch den unnötig gewordenen Graph-Cut erzielt, sondern auch eine Minimierung des Speicherbedarfs herbeigeführt werden.

Ähnlichkeitsanalyse: Sind die Daten zu den Originalaufnahmen, den Kameras sowie das Tiefengitter für die Korrespondenzsuche über die Datenstruktur verfügbar, müssen lediglich noch Felder für das Konvergenzproblem bereitgestellt werden. Um entsprechende Berechnungen durchführen zu können, werden die Normalen der Bildpunkte P sowie der zu P passende optimale Tiefenkandidat aus der letzten Iteration benötigt. Zwar wurde für die Bestimmung des Letzteren bereits ein Feld im Zusammenhang mit dem Tiefengitter angelegt, jedoch wird dieses während der Korrespondenzsuche stets aktualisiert, wodurch relevante Informationen verloren gehen. Für die Konvergenzberechnungen muss also eine Kopie des jeweiligen Feldes erstellt werden. Zusätzlich wird Speicher für die Datenfelder zur Eingrenzung des für P abschätzbaren Tiefenintervalls reserviert.

Verwendung	**Elemente**	**Speicherbedarf (in Bytes)**
Bilddaten	Reziproke Bildpaare	$24\ w\ h\ num$
	Entzerrtes Bildpaar	$24\ w\ h$
	Hilfsbilder (Farbwertinterpolation)	$24\ w\ h$
Kameraeigenschaften	Rotationsmatrix	$72\ num + 36$
	Translationsvektor	$24\ num + 12$
	Intrinsische Koeffizienten	48
	Verzerrungskoeffizienten	32
Tiefengitter	Aktuelle Tiefenebene	$4\ w\ h$
	Minimaler Fehler (*Best Match*)	$4\ w\ h$
	Optimialer Tiefenwert	$4\ w\ h$
Ähnlichkeitsanalyse	Normalenabschätzung	$36\ w\ h$
	Optimialer Tiefenwert (*prevDepth*)	$4\ w\ h$
	Tiefenintervall	$8\ w\ h$

Tabelle 2.1: Speicherverbrauch für die Generierung einer Tiefenkarte, wobei w der Bildbreite, h der Bildhöhe und num der Anzahl von betrachteten reziproken Bildpaaren entspricht.

Der Speicherverbrauch für die in der Tabelle 2.1 aufgelisteten Datenfelder, ist über den gesamten Programmverlauf hinweg konstant und kann nach dem Speichern der Tiefenkarte freigegeben werden. Der Speicher für das Tiefengitter könnte entsprechend der Bildskalierung, bzw. bei der Verdopplung der Bildauflösung zwar dynamisch angepasst werden, allerdings ist mit dem Allozieren von Speicher ein zeitlicher Aufwand verbunden, weshalb der maximale Speicherbedarf direkt zu Beginn beansprucht wird.

2.4.1.7 Ergebnisse zur Generierung von Tiefenkarten

Um die Effizienz des entwickelten Verfahrens testen zu können, wurde eine Menge von reziproken Bildpaaren mit einer Auflösung von 4032×3024 Pixel erstellt, unter anderem für ein Motorrad (siehe Abbildung 2.17), mit insgesamt 34 Bildpaaren, und für eine auf einer Hängematte liegende Frau (siehe Abbildung 2.16), mit insgesamt 53 Bildpaaren. Die für die Generierung der Tiefenkarten notwendigen Berechnungen wurden auf einem Rechner mit einem 3.333 GHz Intel Core i7-980X Prozessor, inklusive einem 6 GB DDR3-1333 Hauptspeicher und einer NVIDIA GeForce GTX 580 Grafikkarte (841/4204 MHz) durchgeführt. Der parallele Algorithmus zur Generierung der Tiefenkarten wurde mithilfe von CUDA implemeniert. Die Anzahl der zu einem Objekt bzw. einer räumlichen Szene produzierten Tiefenkarten entspricht der Anzahl der jeweils aufgenommenen reziproken Bildpaare. Für jede zu konstruierende Tiefenkarte wurden die Daten von fünf verschiedenen reziproken Bildpaaren eingelesen, auf die Grafikkarte übertragen und für die Tiefenabschätzung verwendet. Um die durch den Gebrauch mehrerer reziproker Bildpaare proklamierte Qualitätssteigerung der Rekonstruktionsergebnisse nachvollziehen zu können, wurden bei der Konstruktion des zum Motorrad gehörenden 3D-Modells zusätzlich Tiefenkarten erzeugt, welche auf die Verwendung eines einzelnen reziproken Bildespaares zurückgeführt werden können (siehe Abbildung 2.18).

Die Abbildungen 2.18 bis 2.21 repräsentieren die aus der Tiefenanalyse hervorgehenden Ergebnisse, welche im Folgeschritt für die 3D-Rekonstruktion des Motorrades verwendet wurden. Aus der Abbildung 2.18, welche die zu dem in Abbildung 2.17 dargestellten reziproken Bildpaar korrespondierenden Tiefenkarten beinhaltet, wird ersichtlich, dass die Tiefenabschätzung sowie die Differenzierung zwischen den einzelnen Details der im Bild befindlichen Elemente mit einer steigenden Anzahl von betrachteten Bildpaaren immer präziser wird. Dieser Effekt macht sich vor allem im Bereich des Tanks und innerhalb der Bodenregion (die einzelnen Kacheln sind kaum voneinander unterscheidbar) bemerkbar. Zudem werden Fehlinterpretationen während der Tiefenanalyse und die damit im Zusammenhang stehende Ausprägung von Artefakten vermindert (siehe Abbildung 2.20 und 2.21). Die Verzerrung der Bildkomponenten kommt dadurch zustande, dass die Tiefenkarte aus Sicht der virtuellen Kamera erzeugt wird, die mittig zwischen den auszuwertenden Bildern platziert wird (siehe Abbildung 2.10). Zudem sorgt der Gebrauch von mehreren Bildpaaren dafür, dass während der Berechnungen das Intervall, in welchem sich der optimale Tiefen-

Abbildung 2.16: Ein reziprokes Bildpaar (oben), zu welchem eine Tiefenkarte (unten links) mit den entsprechenden Normalen (unten rechts) generiert wurde.

wert i für einen Bildpunkt P befinden muss, stärker eingegrenzt werden kann. Dadurch konnte die Tiefenanalyse auf ein Minimum an zu betrachtenden Tiefenebenen reduziert und somit die Performanz erhöht werden (siehe Tabelle 2.2). Wurde für die Generierung einer Tiefenkarte nur ein Bildpaar verwendet, konnte diese in durchschnittlich 2 Stunden und 55 Minuten berechnet werden (bzgl. einer Bildauflösung von 4032×3024). Bei der Verwendung von 5 Bildpaaren benötigte das Verfahren hingegen lediglich ca. 1 Stunde und 23 Minuten, was einer Performanzsteigerung von 50 Prozent entspricht. Für die Initialisierung wird aufgrund des Ladens mehrerer Bildpaare und für deren Entzerrung zwar mehr Zeit in Anspruch genommen, allerdings ist dieser Mehraufwand gerade in Hinblick auf den daraus resultierenden Zeitgewinn, welcher während der Korrespondenzanalyse und der Suche nach dem *Best Match* erzielt werden kann, durchaus akzeptabel.

Berechnungsphase	1 Bildpaar	5 Bildpaare
Initialisierung	21.9672s	57.5131s
Korrespondenzsuche/Matching	10514.05s	4968.65s
Punktwolke/Normale	0.0026s	0.0026s
Laufzeit (gesamt)	2h 55m 36s	1h 23m 46s

Tabelle 2.2: Die durchschnittlich benötigte Laufzeit zur Berechnung einer 4032×3024 Pixel großen Tiefenkarte, unter Verwendung von reziproken Bildpaaren.

Abbildung 2.17: Eines der zur Rekonstruktion des Motorrades verwendeten reziproken Bildpaare, linkes Kamerabild (oben) und rechtes Kamerbild (unten).

Abbildung 2.18: Die zu dem reziproken Bildpaar gehörende Tiefenkarte (oben), inklusive der Optimierung durch vier weitere Bildpaare (unten).

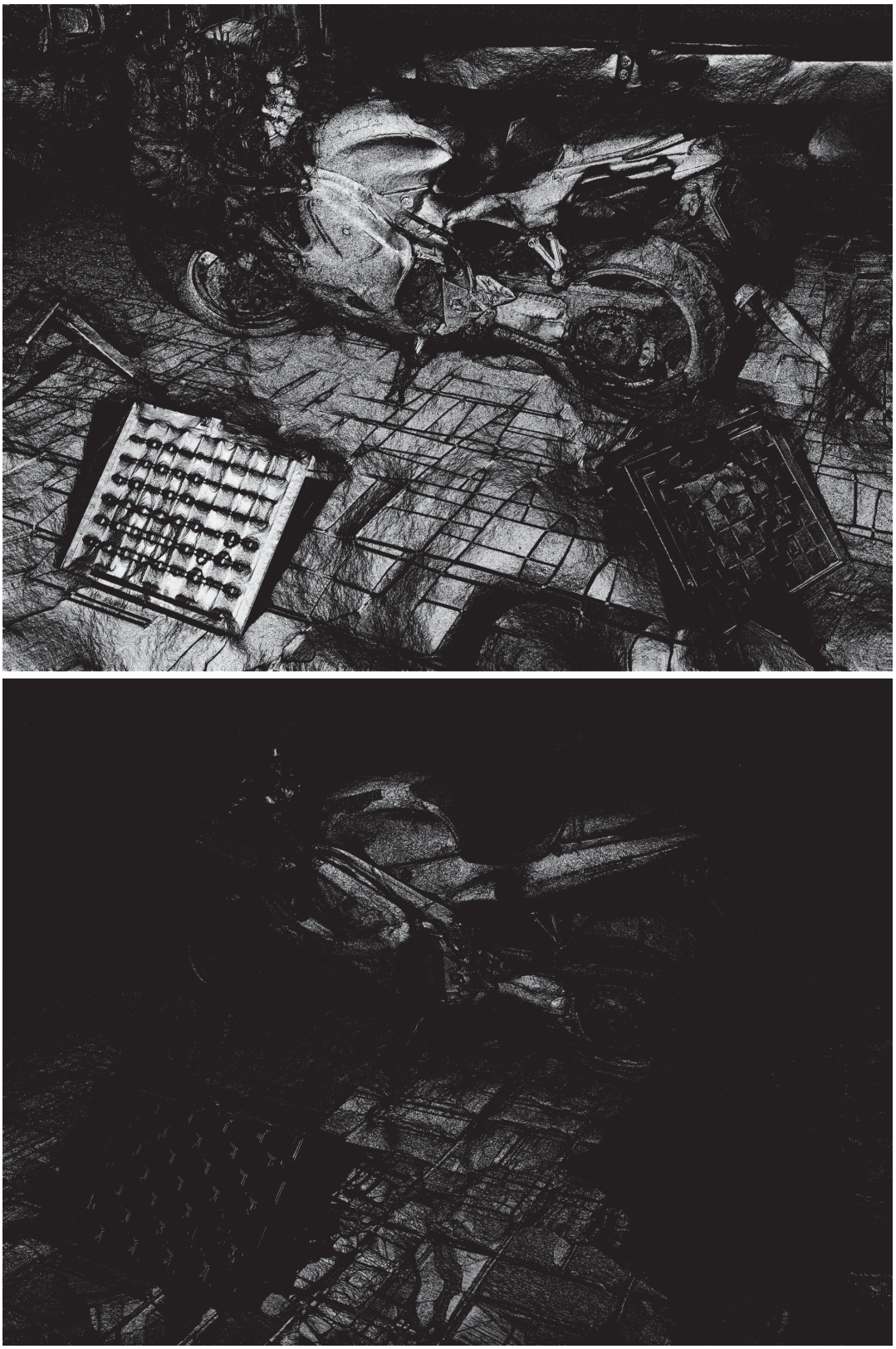


Abbildung 2.19: Darstellung der resultierenden Konfidenzmatrizen, ohne Optimierung (oben) und unter Berücksichtigung weiterer Bildpaare (unten).

Abbildung 2.20: Beispiel einer generierten Tiefenkarte (Mitte), unter Berücksichtigung eines einzelnen reziproken Bildpaares (oben) und die Verbesserung durch vier weitere Paare (unten).

Abbildung 2.21: Beispiel einer generierten Tiefenkarte (Mitte), unter Berücksichtigung eines einzelnen reziproken Bildpaares (oben) und die Verbesserung durch vier weitere Paare (unten).

In Abbildung 2.19 wird die zu den in Abbildung 2.18 gehörende Konfidenz (siehe Abschnitt 2.4.1.4), also die Qualität der vollzogenen Tiefenanalyse dargestellt. Da die Qualität mit steigender Konfidenz zunimmt, ist die Abschätzung der Tiefe umso genauer, je dunkler und plastischer die entsprechenden Bildbereiche ausfallen.

2.4.2 Generierung einer dreidimensionalen Objektansicht

Wurden für sämtliche Bildpunkte aller verfügbaren reziproken Bildpaare die Tiefeninformationen abgeschätzt, kann mithilfe der vorliegenden Daten ein dreidimensionales Polygonnetz generiert werden. Jede Tiefenkarte repräsentiert eine spezifische Ansicht eines 3D-Objektes oder einer nachzubildenen Raumsituation. Jeder darin befindliche Pixel $P(x, y)$ stellt zusammen mit dem für ihn ermittelten Tiefenwert i, welcher über eine Grauwertabstufung kodiert wurde, einen Punkt $P'(x, y, i)$ im $\mathbb{R}^3$ dar. Die Pixel ergeben in der Summe eine *Punktwolke*. Um die Zusammenhänge zwischen den einzelnen Elementen der *Punktwolke* rekonstruieren zu können, sollten die jeweiligen Punktnormalen vorhanden sein. Diese können ebenfalls unter Verwendung einer Farbkodierung in einer separaten Bilddatei abgespeichert werden. Voraussetzung hierfür ist, dass nach der Bestimmung des zu einem Pixel P gehörenden optimalen Tiefenkandidaten eine Normalenabschätzung (siehe Abschnitt 2.4.1.4) durchgeführt wird. Ohne die Normalen kann der Bezug zu den anderen Punkten der dreidimensionalen Objektoberfläche nicht hergestellt werden.

Aufgrund der Tatsache, dass für die Konstruktion eines zusammenhängenden Polygonnetzes eine Vielzahl von effizienten Algorithmen existiert, welche lediglich eine *Punktwolke* und die entsprechenden Normalen als Eingabe benötigen, beschränken sich die eigenen Forschungsleistungen auf die Erzeugung der Tiefenkarten und damit auf die Generierung der für die letzte Rekonstruktionsphase benötigten Eingabedaten. Um ein vollwertiges Programm zur 3D-Rekonstruktion anbieten zu können, beruhen die abschließenden Implementierungsarbeiten bzw. die nun folgenden Erläuterungen auf den Ideen von Michael Kazhdan, Matthew Bolitho und Hugues Hoppe [Kaz05, KBH06], welche mit einer späteren Veröffentlichung ebenfalls die Parallelisierungsmöglichkeiten ihres Verfahrens vorstellten [BKBH09].

Mit der *Poisson Surface Reconstruction* wird die Verkettung orientierter Punkte von Kazhdan et al. [KBH06] als räumliches Randwertproblem aufgefasst, welches mithilfe der *Poisson*-Gleichung gelöst werden kann. Die Grundidee des Verfahrens besteht darin, eine Relation zwischen den Punktnormalen und der zu rekonstruierenden Objektoberfläche zu finden, auf welcher die Bildpunkte angesiedelt sind. Wichtigstes Element stellt dabei die Indikatorfunktion χ dar, über welche die räumlichen Positionsangaben als im Inneren oder außerhalb des Modells befindliche Punkte klassifiziert werden können. Letztlich kann χ als Grenzfunktion aufgefasst werden, durch welche sämtliche vorliegenden Punktdaten angenähert und somit die Modelloberfläche approximiert werden kann. Demnach ist

die Hauptaufgabe des Verfahrens die Bestimmung der Funktion χ. Hierfür müssen die mathematischen Zusammenhänge zwischen den Gradienten von χ, welche auf der Modelloberfläche den Punktnormalen ähneln, und den einzelnen Elementen der Punktwolke analysiert werden. Dieses Problem lässt sich mithilfe der *Poisson*-Gleichung lösen, welche die optimale Approximation zwischen der Divergenz des Gradientenfeldes $\Delta\chi$ und der Divergenz des Vektorenfeldes der Punktnormalen $\nabla \cdot \vec{V}$ repräsentiert:

$$\Delta\chi \equiv \nabla \cdot \nabla\chi = \nabla \cdot \vec{V}. \tag{2.55}$$

Abbildung 2.22 soll die Vorgehensweise der *Poisson Surface Reconstruction* anhand eines zweidimensionalen Beispiels verdeutlichen. Ausgehend von einer Punktmenge inklusive der zugehörigen Normalen wird das Gradientenfeld $\nabla\chi$ abgeschätzt und zur Berechnung der Indikatorfunktion χ verwendet, mit deren Hilfe die Modelloberfläche ∂M rekonstruiert werden kann.

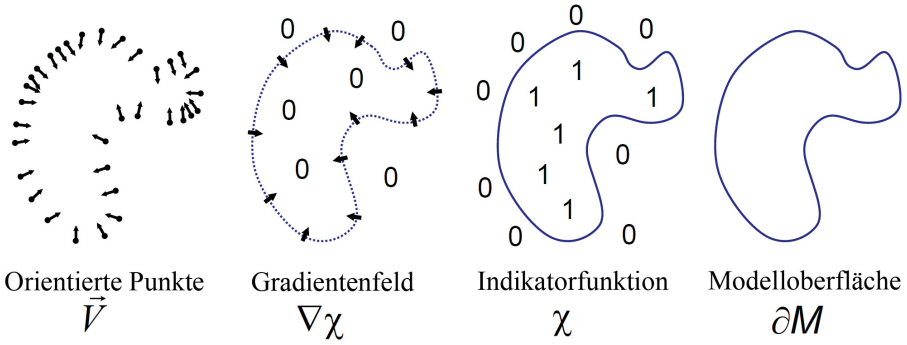


Abbildung 2.22: Schematische Darstellung zur Funktionsweise der *Poisson Surface Reconstruction* in 2D [KBH06].

Der Vorteil des *Poisson*-Ansatzes liegt darin, dass alle Oberflächenpunkte auf einmal berücksichtigt werden, ohne eine Unterteilung der Punktmengen vorzunehmen. Aus diesem Grund ist dieses Verfahren gegenüber Rauscheffekten unempfindlich. Zudem besteht die Möglichkeit, die Berechnungen mittels lokalen Basisfunktionen hierarchisch aufzubauen und zwar derart, dass mit jeder Stufe die Genauigkeit der approximierten Modelloberfläche zunimmt. Somit kann die Komplexität auf ein beschränktes, lineares System reduziert werden. Im Folgenden sollen die verschiedenen Verfahrensschritte erläutert und entsprechende Rahmenbedingungen diskutiert werden.

2.4.2.1 Definition des Gradientenfeldes

Es sei eine Menge von Punkten (auch *Samples* genannt) gegeben, welche der Oberfläche eines Modells M zugehörig sind. Jedem *Sample* s wird neben dem Positionsvektor $\vec{s}$ eine Oberflächennormale $\vec{n}_s$ zugeordnet. Ziel ist es, eine vollständig triangulierte sowie lückenlose Approximation der zu M passenden Modelloberfläche zu generieren. Da jede

Oberfläche eines Objektes M durch eine charakteristische Funktion χ_M formal dargestellt werden kann, ist es für die Rekonstruktion von M völlig ausreichend, das Funktional χ_M zu ermitteln. Mithilfe der Funktion χ_M kann festgestellt werden, ob sich ein beliebiger Punkt P innerhalb oder außerhalb des Modells M befindet:

$$\chi_M(P) = \begin{cases} 0, & \text{wenn } P \text{ außerhalb von } M \\ 1, & \text{wenn } P \text{ innerhalb von } M. \end{cases} \tag{2.56}$$

Aus obiger Definition wird ersichtlich, dass es sich bei χ_M um eine partielle, unstetige Skalarfunktion handelt, durch welche jedem Punkt im $\mathbb{R}^n$ ein Skalar zugeteilt wird. Der Gradient des resultierenden Skalarfeldes kann mithilfe des *Nabla*-Operators ∇ berechnet werden. Dieser repräsentiert einen Vektor, dessen Komponenten partiellen Ableitungskoeffizienten entsprechen. Angewendet auf ein beliebiges Skalarfeld f im $\mathbb{R}^3$ ergibt sich folgender allgemeiner Sachverhalt:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^T \tag{2.57}$$

Der Vektor ∇f ist immer dann ungleich dem Nullvektor, wenn an einem Punkt im $\mathbb{R}^3$ ein Übergang zwischen den in f definierten Skalarwerten zu verzeichnen ist. Da χ_M für einen beliebigen Punkt lediglich die Werte 0 oder 1 annehmen kann und der Wechsel zwischen den beiden Skalaren ausschließlich an den Positionen der gegebenen *Samples* s stattfindet (diese liegen auf der Oberfläche), entspricht die Berechnung des zugehörigen Gradientenfeldes, der Invertierung aller Punktnormalen $\vec{n}_s$:

$$\nabla \chi_M(\vec{s}) = -\vec{n}_s. \tag{2.58}$$

Um eine Näherung für die gesuchte charakteristische Funktion χ_M zu erhalten, genügt es, den *Nabla*-Operator ∇ aufzulösen und zwischen den einzelnen Gradienten an den Positionen $\vec{q}$ zu interpolieren. Eine derartige Interpolation ist jedoch nicht ohne Weiteres möglich, da χ_M eine partiell unstetige Funktion ist. Allerdings kann durch Faltung mithilfe eines Glättungsfilters $\tilde{F}$ die stetige Funktion $\tilde{\chi}_M$ gewonnen werden:

$$\tilde{\chi}_M = \tilde{F} * \chi_M. \tag{2.59}$$

Zusammenfassend kann das zur Approximation der zu einem Objekt M passenden Modelloberfläche ∂M benötigte Gradientenfeld wie folgt definiert werden [KBH06]

$$\nabla(\chi_M * \tilde{F})(\vec{q}) = \int_{\partial M} \tilde{F}_p(\vec{q})\ \vec{n}_p\ dp\ , \tag{2.60}$$

wobei $\vec{p} \in \partial M$ einer Position im Raum $\vec{n}_p$ der Normalen im Punkt $\vec{p}$, $\tilde{F}(\vec{q})$ dem Glätterungsfilter für den Gradienten an der Position $\vec{q}$, $\tilde{F}_p(\vec{q}) = \tilde{F}(\vec{q} - \vec{p})$ einer Translation zum Punkt $\vec{p}$ und χ_M der charakteristischen Funktion von M entspricht. Laut obigem mathematischen Sachverhalt gilt also die Gleichheit zwischen den Gradienten der geglätteten charakteristischen Funktion und dem Vektorfeld, welches durch das Glätten der Oberflächennormalen entsteht.

2.4.2.2 Approximation des Gradientenfeldes

Da derzeit noch keine Informationen zur Geometrie von ∂M verfügbar und die Beziehungen zwischen den einzelnen *Samples* s unklar sind, kann das Oberflächenintegral in der Formel 2.60 noch nicht berechnet werden. Die zu den *Samples* vorliegenden Informationen genügen jedoch, um das Integral mittels einer diskreten Summenbildung anzunähern. Hierfür wird die gegebene Punktmenge S zunächst zur Partitionierung von ∂M in disjunkte Patches $\mathcal{P}_s \subset \partial M$ verwendet, die in der Summe der gesuchten Modelloberfläche entsprechen (siehe Formel 2.61). Im Anschluss wird das Integral über die einzelnen Patches $\mathcal{P}_s$ durch eine Skalierung der *Samples* $s \in S$ auf Patchgröße angenähert (siehe Formel 2.62). Es ergeben sich also folgende mathematische Zusammenhänge:

$$\begin{aligned} \nabla\tilde{\chi}_M(\vec{q}) = \nabla(\chi_M * \tilde{F})(\vec{q}) &= \int_{\partial M} \tilde{F}_p(\vec{q})\ \vec{n}_p\ dp \\ &= \sum_{s \in S} \int_{\mathcal{P}_s} \tilde{F}_p(\vec{q})\ \vec{n}_p\ dp \qquad (2.61) \\ &\approx \sum_{s \in S} |\mathcal{P}_s|\ \tilde{F}_s(\vec{q})\ \vec{n}_s \equiv \vec{V}(\vec{q})\ . \qquad (2.62) \end{aligned}$$

Während der Skalierung gehen Informationen zur Geometrie verloren, was zu Fehlern im rekonstruierten Modell führt. Um diese Fehler möglichst gering zu halten, sollte das Sampling mit einer entsprechend hohen Auflösung erfolgen und ein geeigneter Glättungsfilter verwendet werden.

2.4.2.3 Lösen des Poisson-Problems

Mit der Definition des Vektorfeldes $\vec{V}$ (siehe Formel 2.62) kann nun die stetige charakteristische Funktion $\tilde{\chi}_M$ ermittelt werden, indem die Gleichung $\nabla\tilde{\chi}_M = \vec{V}$ gelöst wird. Da laut Kazhdan et al. [KBH06] die Approximation einer Modelloberfläche ∂M auf die Lösung des *Poisson*-Problems zurückgeführt werden kann (siehe Formel 2.55), muss hierfür der *Nabla*-Operator ∇ aus obiger Gleichung entfernt werden. Dies kann durch eine erneute Anwendung von ∇ bewerkstelligt werden, wodurch sich das folgende numerische Kriterium zur Approximation von $\tilde{\chi}_M$ formulieren lässt:

$$\Delta\tilde{\chi}_M = \nabla \cdot \vec{V}. \qquad (2.63)$$

In Hinblick auf die Implementierung muss zunächst in Abhängigkeit von den *Samples* $s \in S$ ein Funktionenraum $\mathbb{F}_{\mathcal{O}}$ definiert werden, in welchem das geschilderte Problem diskretisiert werden kann. Aufgrund der Diskretisierung kann das Vektorfeld $\vec{V}$ als Summe der impliziten Funktionen angesehen und damit die *Poisson*-Gleichung gelöst werden. Um über eine geeignete Repräsentation der Funktionen zu verfügen und das *Poisson*-Problem in effizienter Form beheben zu können, favorisierten Kazhdan et al. [KBH06] die Verwendung eines adaptiven Octree.

Für die Rekonstruktion der Modelloberfläche werden alle *Samples* in ein Octree $\mathcal{O}$ der Tiefe $\mathcal{D}$ eingefügt und zwar so, dass auf der Tiefenebene $\mathcal{D}$ jeder Blattknoten genau einen Punkt s enthält. Im Octree sind demnach maximal $2^{n\mathcal{D}}$ Knoten enthalten, wobei n der Dimension des betrachteten Raumes entspricht. Jedem Blatt $o \in \mathcal{O}$ wird eine Funktion $\mathcal{F}_o \in \mathbb{F}_{\mathcal{O}}$ zugeordnet, welche sich an der Stelle $\vec{q}$, im Bezug auf die Basisfunktion $\mathcal{F}$, mittels einer Translation hin zum Zentrum c eines Blattknotens o und einer Skalierung, unter Berücksichtigung der Knotenbreite w, bestimmen lässt:

$$\mathcal{F}_o(\vec{q}) = \mathcal{F}\left(\frac{\vec{q}-c}{w}\right)\frac{1}{w^3}. \tag{2.64}$$

Die Basisfunktion $\mathcal{F}$ ist durch folgende Definition gegeben und entspricht einer n-fachen Faltung im $\mathbb{R}^3$ mit sich selbst:

$$\mathcal{F}(\vec{q}) \equiv (\mathcal{B}(p.x)\ \mathcal{B}(p.y)\ \mathcal{B}(p.z))^{*n} \text{ mit } \mathcal{B}(t) = \begin{cases} 1, & \text{wenn } |t| < 0.5 \\ 0, & \text{sonst.} \end{cases} \tag{2.65}$$

Mithilfe der obigen Definitionen können die Gradienten approximiert und somit das Vektorfeld $\vec{V}$ als lineare Summe der Blattfunktionen $\mathcal{F}_o$ dargestellt werden:

$$\vec{V}(\vec{q}) = \sum_{s\in S}\sum_{o\in\mathcal{O}} \mathcal{F}_o(\vec{q})\ \vec{n}_s. \tag{2.66}$$

Die innere Summe muss jedoch nicht über alle Knoten des Octrees laufen. Die Anzahl der betrachteten Knoten kann auf die direkten Nachbarknoten $N_{\mathcal{D}}(s)$ von o begrenzt werden, da für alle anderen die Basisfunktion $\mathcal{F}$ den Wert 0 liefert. Da die direkten Nachbarn einen unterschiedlichen Einfluss auf den zu berechnenden Gradienten an der Position $\vec{q}$ ausüben, werden diese in Abhängigkeit der *Samples* $\alpha_o(s)$ gewichtet:

$$\vec{V}(\vec{q}) = \sum_{s\in S}\sum_{o\in N_{\mathcal{D}}(s)} \alpha_o(s)\ \mathcal{F}_o(\vec{q})\ \vec{n}_s. \tag{2.67}$$

Mit der Definition des Vektorfeldes $\vec{V}$ muss nur noch die *Poisson*-Gleichung gelöst werden. Hierbei könnten jedoch Komplikationen auftreten, da der Funktionenraum $\mathbb{F}_{\mathcal{O}}$ unter Berücksichtigung von $\nabla\tilde{\chi}_M$ bzw. $\vec{V}$ konstruiert wurde, jedoch nicht für $\Delta\tilde{\chi}_M$ und $\nabla\cdot\vec{V}$ gelten muss. Um derartige Fälle zu vermeiden, werden die entsprechenden Daten nach

$\mathbb{F}_{\mathcal{O}}$ projiziert. Da es sich bei der *Poisson*-Gleichung um eine partielle Differentialgleichung zweiten Grades handelt, kann diese als Minimierungsproblem zur Berechnung der kleinsten Quadrate (engl. *least-squares*) umformuliert werden. In Verbindung mit der Projektion nach $\mathbb{F}_{\mathcal{O}}$ ergibt sich folgender mathematischer Zusammenhang:

$$\min \sum_{o \in \mathcal{O}} \left\| \langle \Delta \tilde{\chi}_M - \nabla \cdot \vec{V}, \mathcal{F}_o \rangle \right\|^2 = \min \sum_{o \in \mathcal{O}} \left\| \langle \Delta \tilde{\chi}_M, \mathcal{F}_o \rangle - \langle \nabla \cdot \vec{V}, \mathcal{F}_o \rangle \right\|^2. \tag{2.68}$$

Sei $\vec{v}$ ein $|\mathcal{O}|$-dimensionaler Vektor, dessen Koeffizienten an der Position o den Werten von $v_o = \langle \nabla \cdot \vec{V}, \mathcal{F}_o \rangle$ entsprechen und als bekannt angesehen werden können. Weiterhin gilt $\tilde{\chi}_M = \sum_{o \in \mathcal{O}} x_o \mathcal{F}_o$, wodurch zur Lösung des *Poisson*-Problems der Vektor $\vec{x} \in \mathbb{R}^{|\mathcal{O}|}$ gesucht wird, mit dessen Hilfe die Projektion von $\Delta \tilde{\chi}_M$, über die einzelnen Funktionen $\mathcal{F}_o$ hinweg, der von $\vec{v}$ angenähert werden kann. Werden die *Laplace*-Operatoren von Δ in Verbindung mit der Projektion nach $\mathbb{F}_{\mathcal{O}}$ in einer $|\mathcal{O}| \times |\mathcal{O}|$-Matrix $\mathcal{L}$ zusammengefasst, kann das Minimierungsproblem (siehe Formel 2.68) in folgende Matrixschreibweise überführt werden:

$$\min \sum_{\vec{x} \in \mathbb{R}^{|\mathcal{O}|}} \left\| \mathcal{L}\vec{x} - \vec{v} \right\|^2, \tag{2.69}$$

wobei $\mathcal{L}\vec{x}$ das Skalarprodukt zwischen den *Laplace*-Operatoren und den einzelnen Funktionen $\mathcal{F}_o$ repräsentiert und die in der Matrix $\mathcal{L}$ befindlichen Einträge an der Position (o, o') definiert sind durch:

$$\mathcal{L}_{(o,o')} \equiv \left\langle \frac{\partial^2 \mathcal{F}_o}{\partial x^2}, \mathcal{F}_{o'} \right\rangle + \left\langle \frac{\partial^2 \mathcal{F}_o}{\partial y^2}, \mathcal{F}_{o'} \right\rangle + \left\langle \frac{\partial^2 \mathcal{F}_o}{\partial z^2}, \mathcal{F}_{o'} \right\rangle. \tag{2.70}$$

Das aus der Formel 2.69 resultierende Gleichungssystem kann nun mithilfe eines *Newton*-Verfahrens oder mit der Methode zur Bestimmung der konjugierten Gradienten (engl. *conjugate gradient solver* - CG) gelöst werden. Nachdem diese Berechnungen erfolgt sind, kann die Modelloberfläche rekonstruiert werden. Die entsprechende Vorgehensweise wird im folgenden Kapitel näher erläutert.

An dieser Stelle sei abschließend vermerkt, dass sich die hier aufgeführten Erläuterungen auf das *uniforme Sampling* beziehen. Um das *nicht-uniforme Sampling* zu unterstützen, muss die Definition des Vektorfeldes $\vec{V}$ (siehe Formel 2.67) verallgemeinert werden, indem die innere Summe über das Verhältnis zwischen einem *Sample* s und den einzelnen durch ihn approximierten Punkten der Modelloberfläche gewichtet wird. Diese Gewichtung muss zudem bei der Auflösung des Minimierungsproblems berücksichtigt werden. Für eine genauere Erläuterung sei auf die entsprechende Literatur verwiesen [KBH06].

2.4.2.4 Extraktion der Modelloberfläche

Nachdem nun die charakteristische Funktion $\tilde{\chi}_M$ zu einem Modell M ermittelt wurde, muss diese letztlich noch trianguliert werden, um die approximierte Modelloberfläche ∂M zu erhalten. Dies wird mithilfe des konstruierten Octrees $\mathcal{O}$ und dem *Marching Cube* Verfahren realisiert.

Ganz allgemein wird bei einem *Marching Cube*-Verfahren ein gegebenes Voxelmodell in mehrere kleinere Würfel (engl. *cubes*) unterteilt, die im Anschluss durchlaufen werden. Während dieses Prozesses wird für jeden Würfel anhand der darin befindlichen Voxel bzgl. der zum Modell gehörenden charakteristischen Funktion die Fläche berechnet, welche als Bestandteil der zu rekonstruierenden Modelloberfläche den jeweiligen Würfel durchtrennt. Somit wird die gesuchte Oberfläche schrittweise zusammengesetzt.

In der Regel erhält der Algorithmus ein reguläres Gitter als Eingabe. Da jedoch zur Diskretisierung ein adaptiver Octree genutzt wurde, liegen die Daten in einer hierarchisch angeordneten Datenstruktur vor, was zu Fehlern bei der Triangulierung führen kann. Problematisch sind dabei die Übergänge zwischen zwei benachbarten Würfeln, deren Detailstufen nicht übereinstimmen. Laut dem Konstruktionsprinzip des verwendeten Octrees wird ein darin enthaltener Würfel solange unterteilt, bis dieser nur noch ein *Sample* in sich einschließt. Da die einzelnen Zellen des Octrees eine unterschiedliche Anzahl von *Samples* aufweisen können, werden diese auch dementsprechend oft unterteilt. Um dieses Problem zu lösen, wird der *Marching Cube*-Algorithmus von der feinsten Detailstufe ausgehend bis zum gröbsten Detailgrad durchlaufen. In jeder Stufe werden die zugehörigen Schnittflächen ermittelt, kombiniert und an die nächste weitergereicht. Für nähere Angaben sei auf die Literatur verwiesen [KKDH07, LC87, SFYC96, WVG92, WKE99].

2.5 Ergebnisse

Das entwickelte Verfahren stellt eines der ersten GPU-Implementierungen im Bereich der 3D-Rekonstruktion dar, welches mithilfe von reziproken Bildpaaren ein dreidimensionales Abbild der in den Kamerabildern dargestellten Szene erzeugt. Bei vergleichbaren Rekonstruktionsverfahren wie beispielsweise der Ansatz von Zickler et al. [ZBK02, ZHK*03] werden die reziproken Bildpaare unter kontrollierten Bedingungen erstellt. Dabei werden die zu rekonstruierenden Modelle innerhalb einer Sphere platziert, auf welcher mehrere Kameras montiert wurden (vgl. Abbildung 2.23). Durch dieses Arrangement ergeben sich mehrere Vorteile. Einerseits kann durch den Aufbau der Abstand sowie die Orientierung der Kameras zum Objekt

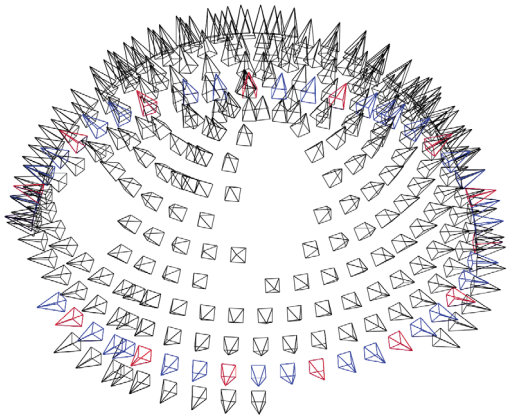


Abbildung 2.23: Schematische Darstellung einer mit 317 Kameras bestückten Sphere, zur Aufnahme von reziproken Bildpaaren [SCD*06].

hin gesteuert werden, was neben der Extraktion der Kalibrierungsdaten für die virtuelle Kamera die Tiefenabschätzung enorm erleichtert, da das Tiefenintervall durch den Aufbau definiert ist und zum anderen kann ein monochromer Hintergrund gewählt werden, sodass die durch störende im Hintergrund befindliche Gegenstände hervorgerufenen Fehlinterpretationen während der Korrespondenzsuche vermieden werden. Zudem werden für die Rekonstruktion meist Objekte mit groben Strukturen und einer matten Oberfläche verwendet, um Fehler ausschließen zu können, die z.B. durch Reflektionen der Umgebung oder aufgrund von hellen Reflektionspunkten verursacht werden.

Bei der Realisierung des im Rahmen dieser Arbeit entwickelten Verfahrens wurde bewusst auf derartige optimale Rahmenbedingungen verzichtet, um ein Programm zur Verfügung zu stellen, welches auch unter Normalbedingungen gute Resultate liefert. Dies hat zur Folge, dass mehrere Freiheitsgrade, wie z.B. der Abstand zur Kamera oder deren Ausrichtung, bei den Berechnungen berücksichtigt und auf Grundlage der Bilddaten ausgewertet werden müssen. Für die Aufnahme der für die Tests benötigten reziproken Bildpaare wurden lediglich zwei Stative, auf welchen die beiden Kameras bzw. die Lichtquelle positioniert wurden, sowie sechs *Targets*, für die Bestimmung der Kalibrierungsdaten (siehe Kapitel 2.1.1.6), verwendet. Für die Kamerakalibrierung wurde eine Methode implementiert, welche die zu den in den einzelnen Kamerabildern befindlichen *SURF-Features* (engl. *speeded up robust features*) gehörenden Positionsangaben extrahiert. Diese sind für die Bestimmung der Kalibrierungsdaten notwendig. Als *SURF-Features* sind hier die innerhalb der auf den *Targets* angebrachten Schachbrettmuster sichtbaren und markanten Kreuzungspunkte zu verstehen (vgl. Abbildung 2.17). In Hinblick auf die Tests wurden Gegenstände oder Raumsituationen gewählt, die eine Vielzahl von feinen Strukuren sowie glänzende und spiegelnde Oberflächen enthalten. Auf einen einheitlichen Hintergrund wurde ebenfalls verzichtet.

Trotz der verallgemeinerten Betrachtungsweise konnten im Bezug auf verwandte Arbeiten mit dem hier vorgestellten Verfahren zur 3D-Rekonstruktion vergleichbare und hinsichtlich der Berücksichtigung spiegelnder Flächen bessere Ergebnisse konstruiert werden. Zudem konnte aufgrund der effizienten parallelen Datenstruktur ein Performanzanstieg gegenüber anderen CPU- und GPU-Implementierungen erzielt werden. Die im Folgenden präsentierten Resultate entsprechenen demnach dem aktuellen Forschungsstand. Wie in Kapitel 2.4.1.7 beschrieben wurde für die Rekonstruktion eine Menge von reziproken Bildpaaren erstellt, welche zur Generierung der Tiefenkarten mit einer Auflösung von 4032×3024 Pixel verwendet wurden. Mithilfe der Tiefenkarten konnte zu jedem Modell eine Punktwolke erzeugt werden, welche als Eingabe für die durchzuführende *Poisson Surface Reconstruction* dient. Die Berechnungen wurden dabei auf einem Rechner mit einem 3.333 GHz Intel Core i7-980X Prozessor, inklusive einem 6 GB DDR3-1333 Hauptspeicher und einer NVIDIA GeForce GTX 580 Grafikkarte (841/4204 MHz) durchgeführt.

Die Abbildungen 2.24 bis 2.28 stellen die Ergebnisse zur 3D-Rekonstruktion des Motorrades und die Abbildungen 2.29 bis 2.31 die zu der in einer Hängematte liegenden Frau

dar. Aufgrund der rekonstruierten dreidimensionalen Modelle können folgende Schlussfolgerungen gezogen werden:

Anzahl verwendeter reziproker Bildpaare: Durch eine höhere Anzahl an verwendeten reziproken Bildpaaren können Fehlinterpretationen während der Tiefenabschätzung reduziert werden. Dies führt dazu, dass die einzelnen Elemente eines zu rekonstruierenden Objektes voneinander unterschieden und somit der optimale Tiefenwert für jeden Bildpunkt gefunden werden kann. Die Präzision der Korrespondenzsuche steigt demnach mit der Anzahl an berücksichtigten Bildpaaren (siehe Kapitel 2.4.1.7), was sich wiederum nachhaltig auf die Rekonstruktionsergebnisse auswirkt. Auf das Modell des Motorrades bezugnehmend wird ersichtlich, dass bei dem Erstellen einer Tiefenkarte, für welche lediglich ein reziprokes Bildpaar ausgewertet wurde, im Endresultat bis auf die grobe Ausrichtung und Position des Objektes zwischen den einzelnen Objektstrukturen kaum bis gar nicht differenziert werden kann (siehe Abbildung 2.24 und 2.25).

Größe und Qualität der Bildaufnahmen: Anhand der Rekonstruktionsergebnisse zeigt sich, dass für eine korrekte Remodellierung einzelner Objekte die Anzahl der aufgenommenen Bilder entscheidend ist. Wird ein Objekt aus mehreren Kamerapositionen und aus verschiedenen Winkeln heraus abgelichtet, können einzelne Strukturen genauer erfasst werden. Da aufgrund von Überschneidungen und Objektüberlagerungen keine korrekte Zuordnung zwischen zwei korrespondierenden Bildpunkten erfolgen kann, werden Teile eines Modells miteinander kombiniert oder sie verschmelzen teilweise mit anderen in der näheren Umgebung befindlichen Objekten. Dieser Effekt wird durch eine schlechte Ausleuchtung der Szenerie verstärkt, da bei der auf Farbwerten oder Farbintensitäten beruhenden Tiefenanalyse das Aufkommen von Mehrdeutigkeiten nicht vermieden werden kann. In Abbildung 2.26 (unten) wird dieser Umstand vor allem im Bereich unterhalb des Motorades erkennbar, welches an dieser Stelle fließend in den Boden übergeht.

Punktwolke und komplettes Modell: Die soeben geschilderte Problematik ist jedoch nicht allein von der Anzahl der vorhandenen Bildpaare oder von deren Qualität abhängig. Der Übergang von der Punktwolke hin zum geschlossenen Polygonnetz ist ebenfalls problematisch. Da die zu rekonstruierende Modelloberfläche mithilfe einer Indikatorfunktion (siehe Kapitel 2.4.2) interpoliert wird, sind die Abstände zwischen den Elementen der Punktwolke entscheidend. Je näher die Punkte beieinander liegen, umso genauer können die Strukturen remodelliert werden. Punkte, die weiter voneinander entfernt sind, werden entweder fälschlicherweise durch eine konvexe bzw. konkave Fläche miteinander verbunden oder bilden eine eigenständige, von dem Modell losgelöste Einheit. Bei der Rekonstruktion des Motorrades kann dies vorwiegend im Bereich des Benzintanks und des Fahrersitzes beobachtet werden (siehe Abbildung 2.26 bis 2.28).

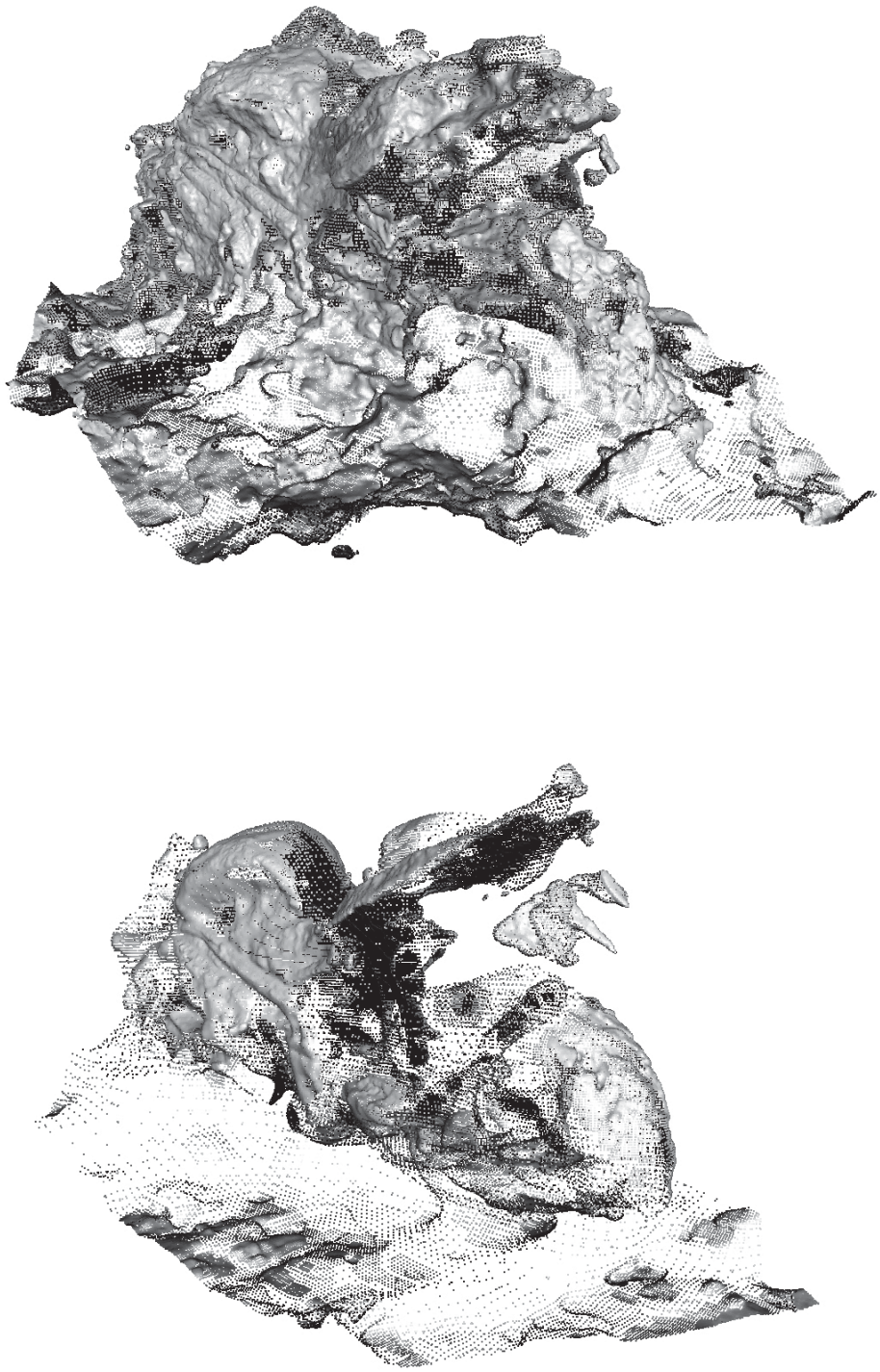


Abbildung 2.24: Darstellung der generierten Punktwolke zur Rekonstruktion des Motorrades, unter Verwendung eines reziproken Bildpaares pro Tiefenkarte (oben) und vier weiteren Paaren (unten).

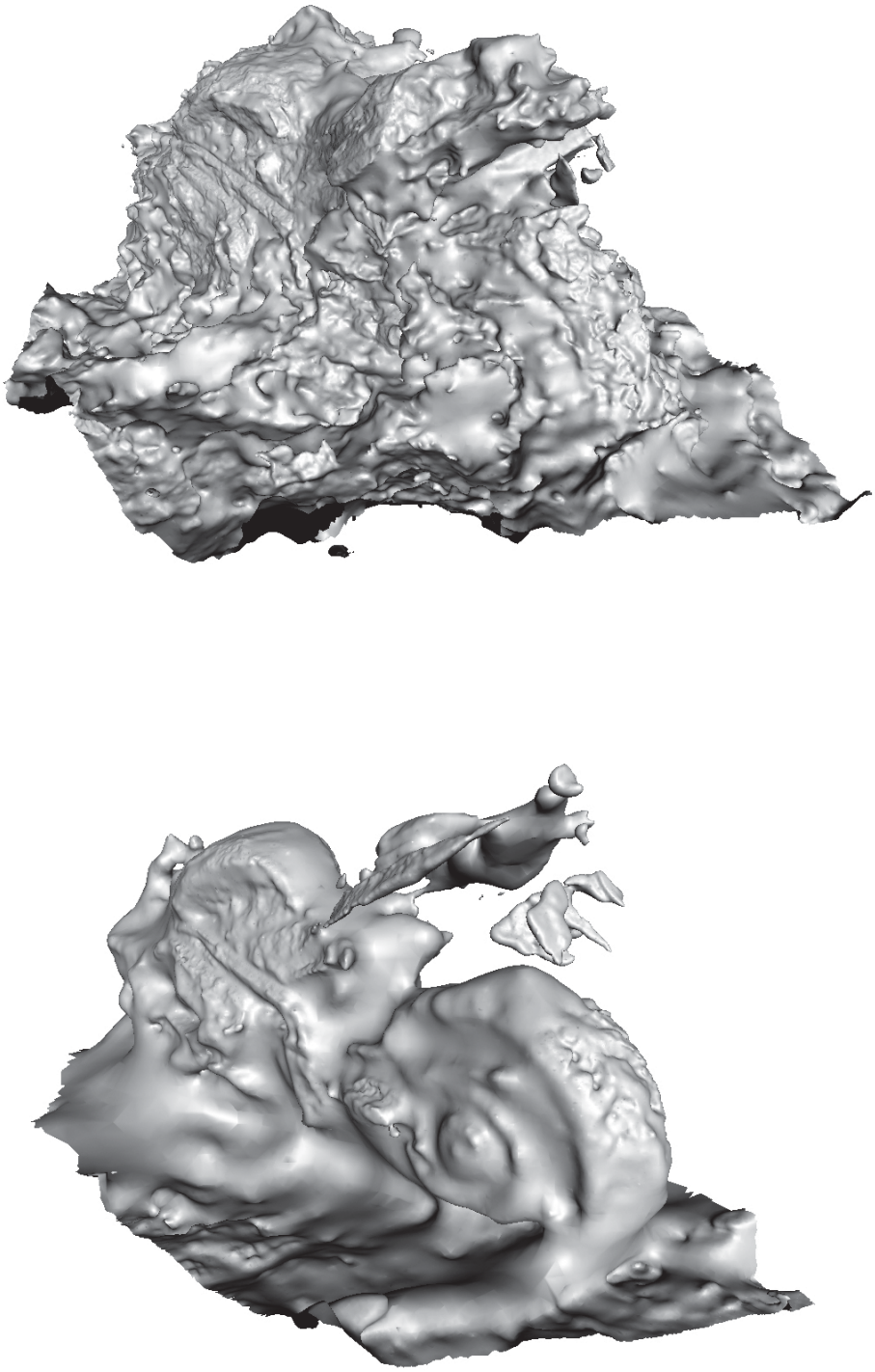


Abbildung 2.25: Darstellung des rekonstruierten Motorrades, mithilfe von einem reziproken Bildpaar pro Tiefenkarte (oben) und unter Berücksichtigung von fünf Bildpaaren (unten).

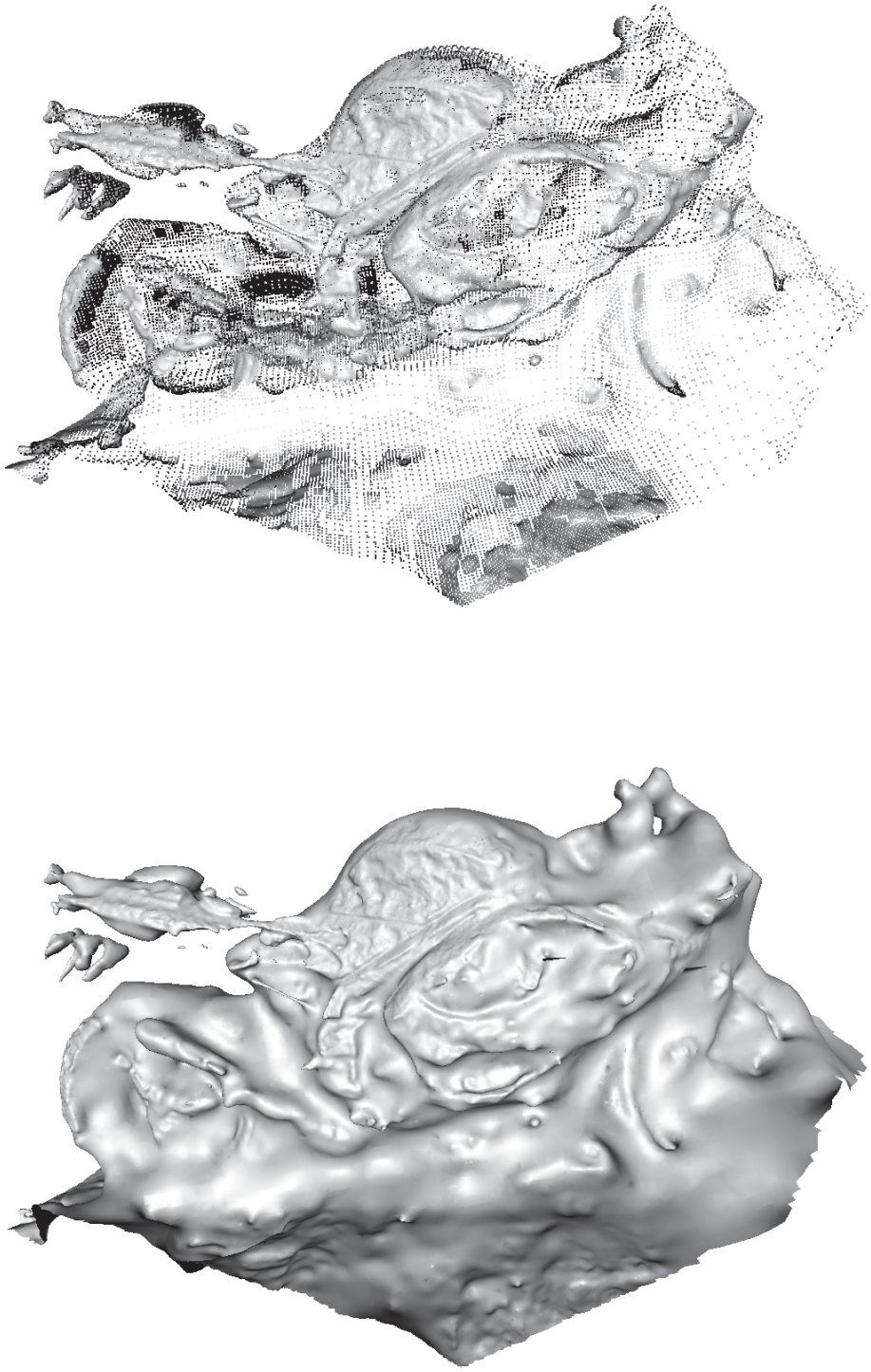


Abbildung 2.26: Darstellung der aus der Tiefenanalyse hervorgehenden Punktwolke (oben), unter Verwendung von fünf reziproken Bildpaaren pro Tiefenkarte, inklusive der interpolierten Modelloberfläche (unten).

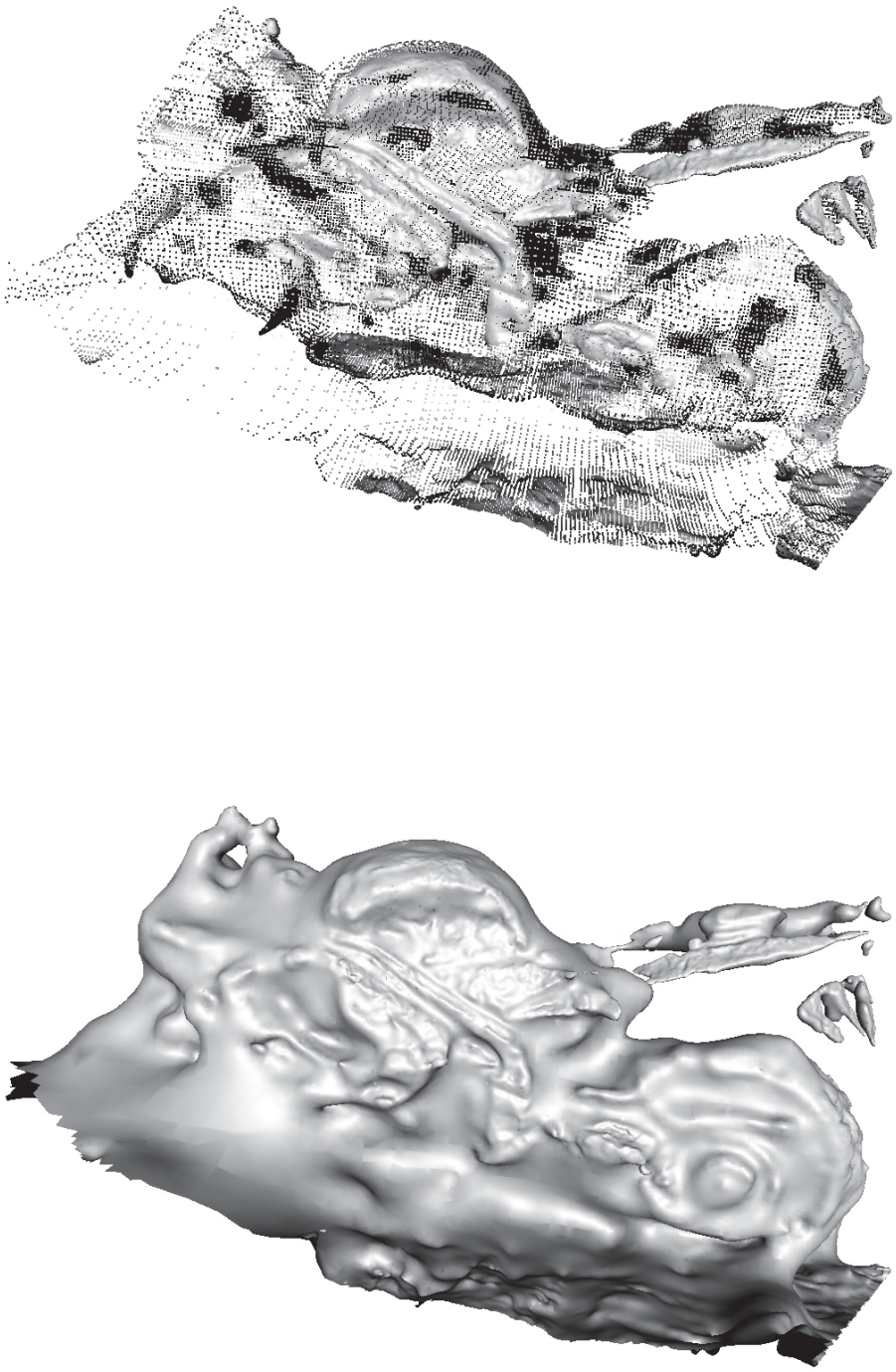


Abbildung 2.27: Darstellung der aus der Tiefenanalyse hervorgehenden Punktwolke (oben), unter Verwendung von fünf reziproken Bildpaaren pro Tiefenkarte, inklusive der interpolierten Modelloberfläche (unten).

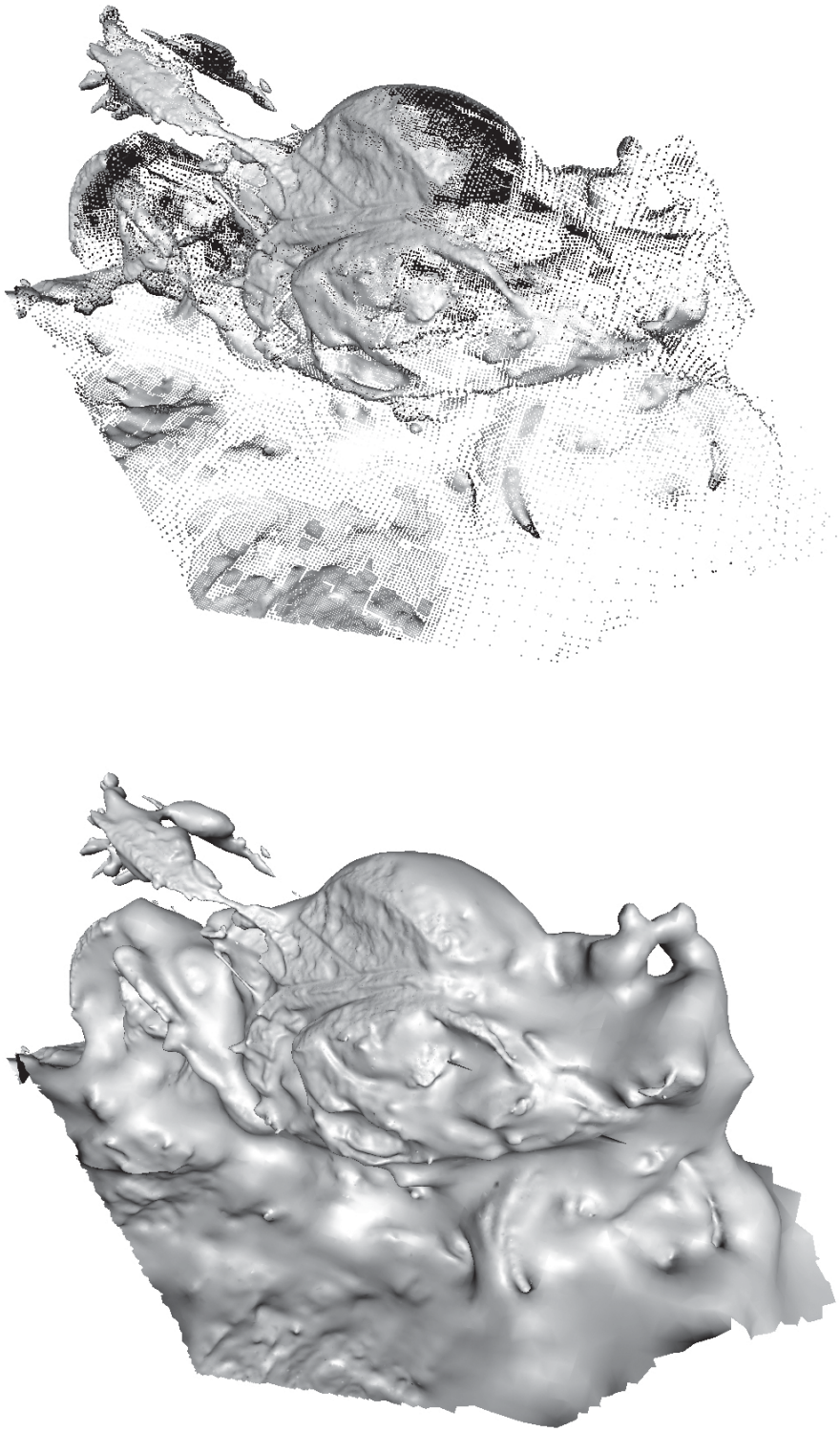


Abbildung 2.28: Darstellung der aus der Tiefenanalyse hervorgehenden Punktwolke (oben), unter Verwendung von fünf reziproken Bildpaaren pro Tiefenkarte, inklusive der interpolierten Modelloberfläche (unten).

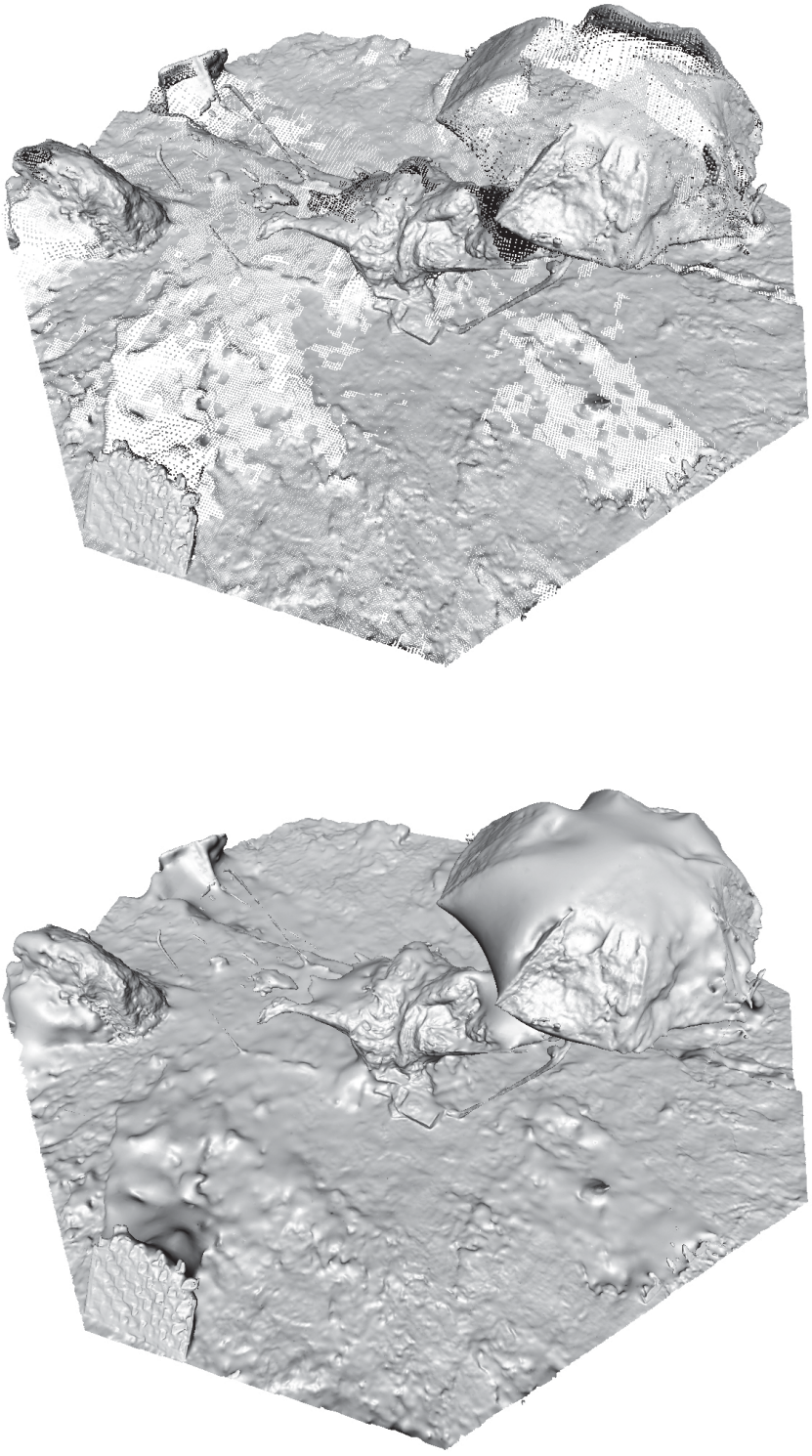


Abbildung 2.29: Rekonstruiertes Modell der in der Hängematte liegenden Frau (unten), inklusive der aus der Tiefenanalyse resultierenden Punktwolke (oben).

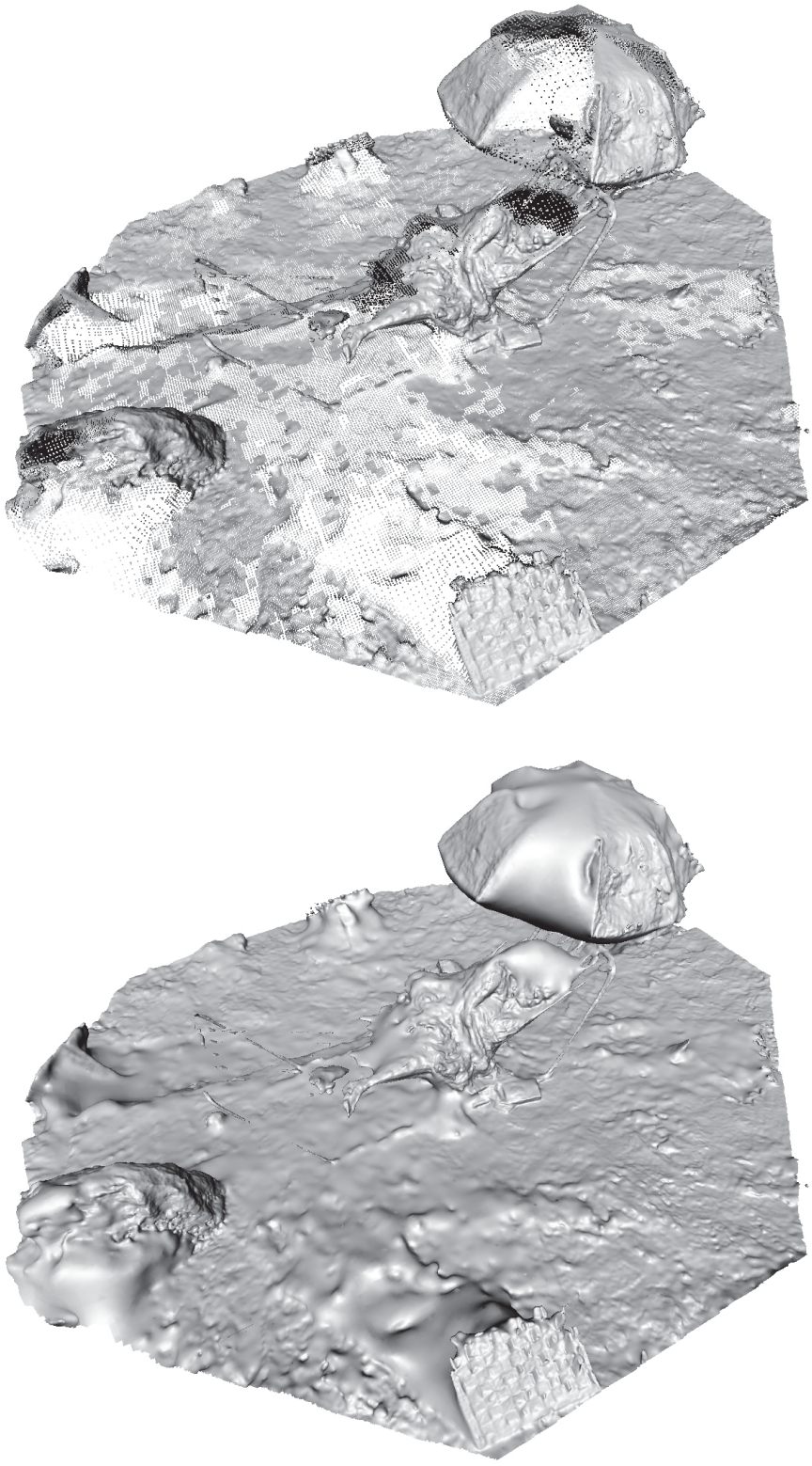


Abbildung 2.30: Rekonstruiertes Modell der in der Hängematte liegenden Frau (unten), inklusive der aus der Tiefenanalyse resultierenden Punktwolke (oben).

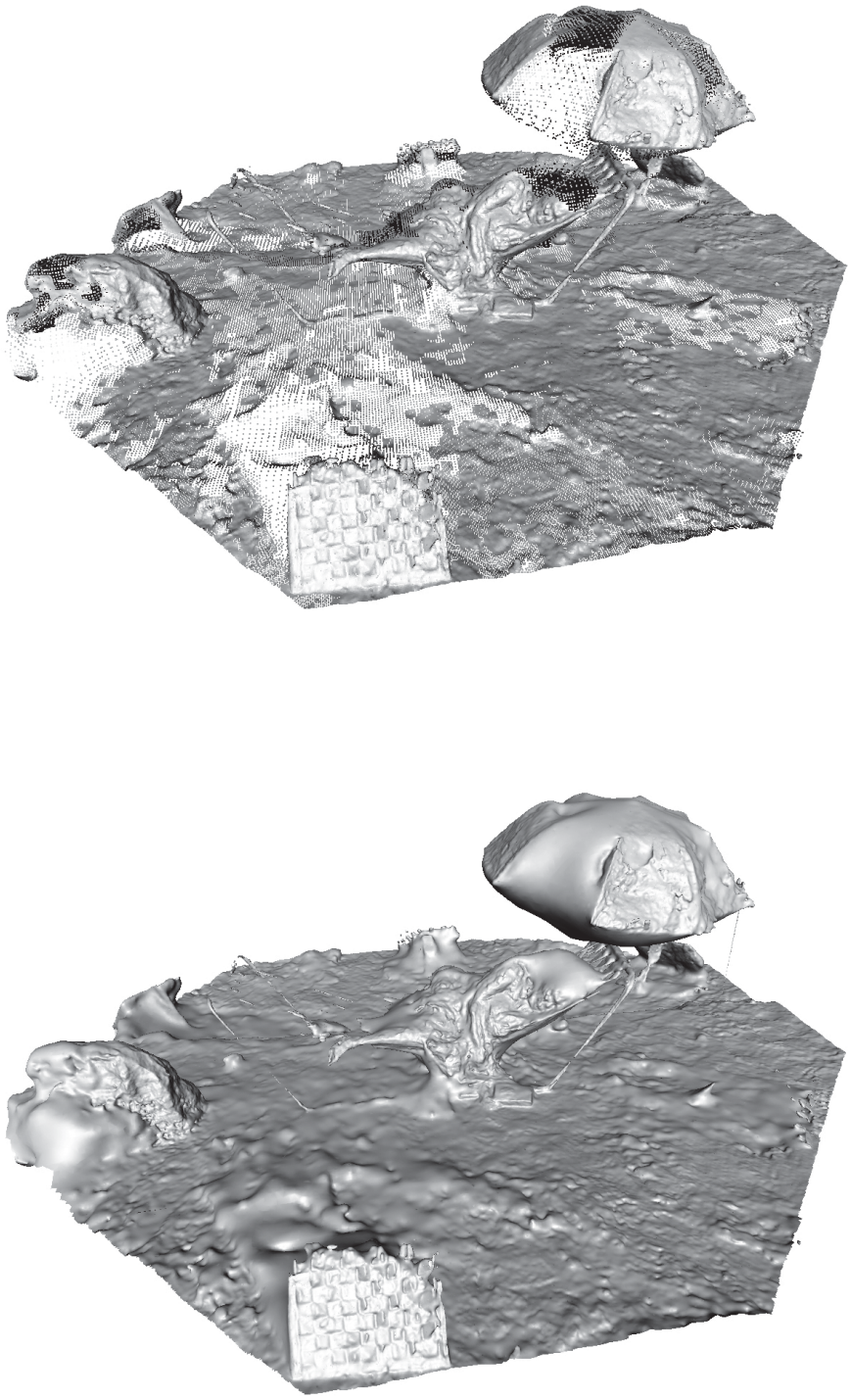


Abbildung 2.31: Rekonstruiertes Modell der in der Hängematte liegenden Frau (unten), inklusive der aus der Tiefenanalyse resultierenden Punktwolke (oben).

2.6 Zusammenfassung

In den letzten Kapiteln wurde ein Verfahren zur 3D-Rekonstruktion diskutiert, welches reziproke Bildpaare verarbeitet, um Korrespondenzen zwischen den in den Einzelaufnahmen befindlichen Objekten aufzudecken und die daraus resultierenden Tiefeninformationen zur Generierung eines dreidimensionalen Abbildes der durch die Bilder wiedergegebenen räumlichen Szene zu nutzen. Für die Tiefenanalyse wurde eine virtuelle Kamera definiert, aus deren Sicht die Tiefenkarten erzeugt werden. Um die zu einem Bildpunkt P gehörenden Tiefenkandidaten i bewerten und miteinander vergleichen zu können, wurde ein dreidimensionales Tiefengitter deklariert, in welchem zu jedem Punkt P der Ähnlichkeitsgrad zwischen den aus den Eingabebildern bzgl. einer hypothetisch angenommen Tiefe i extrahierten Bildmerkmalen zwischengespeichert wird. Nachdem die Korrespondenzanalyse vollzogen wurde, kann aus dem 3D-Gitter der optimale Tiefenwert abgelesen und in einer Tiefenkarte abgelegt werden. Da bei der Bestimmung des zu P passenden, optimalen Tiefenkandidaten, dessen Nachbarschaftsbeziehungen mitberücksichtigt werden sollten, wurde auf das Tiefengitter ein Graph-Cut-Algorithmus angewendet, um somit den *minimalen Schnitt* innerhalb des als Graphen interpretierten 3D-Gitters bzw. das globale Optimum für P zu finden. Hierbei wurde zunächst die Möglichkeit in Betracht gezogen das Problem auf die Minimierung der L1-Norm zurückzuführen, welche letztlich aus Gründen der Effizienz und zur Steigerung der Performanz durch eine parallele Variante der von Andrew V. Goldberg und Robert E. Tarjan entwickelten *Push-Relabel*-Methode [GT88] ersetzt wurde. Da jedoch die Komplexität des Graph-Cut-Verfahrens in Abhängigkeit zur Dimension des 3D-Gitters zunimmt und sich die damit in Verbindung stehenden Berechnungsprozesse maßgeblich auf die Laufzeit des Programmes auswirken, wurde für die Suche nach dem *Best Match* der *Push-Relabel*-Ansatz verworfen und eine abgewandelte Form der *Belief Propagation* gewählt, welche sich in vielerei Hinsicht als vorteilhaft herausstellte. Durch die verbesserte Herangehensweise wurde die Verwaltung der innerhalb eines dreidimensionalen Tiefengitters gesicherten Ähnlichkeitswerte irrelevant, wodurch das 3D-Gitter aus der Datenstruktur entfernt und der Speicherbedarf signifikant gesenkt werden konnte. Des Weiteren konnte die Suche nach dem optimalen Tiefenwert für P mit der Korrespondenzanalyse verknüpft werden, sodass zum einen die einzelnen Verfahrensschritte effizienter umgesetzt werden konnten und zum anderen für jeden Bildpunkt P die Abschätzung eines Tiefenintervalls möglich wurde, in welchem sich das gesuchte Optimum befinden muss. Aufgrund der Reduktion des Speicherbedarfs konnte in Verbindung mit der innovativen Realisierung notwendiger und auf ein Minimum reduzierbarer Berechnungen ein enormer Performanzanstieg erzielt werden. Zusätzlich konnte der Nachweis erbracht werden, dass sich die Verwendung mehrerer reziproker Bildpaare positiv auf die Tiefenanalyse und somit nachhaltig auf die Rekonstruktionsergebnisse auswirkt. Wurden alle relevanten Tiefenkarten generiert, können mit ihrer Hilfe Punktwolken erzeugt und mittels der *Poisson Surface Reconstruction* die zugehörigen Modelloberflächen nachgebildet werden.

Der vorgestellte Rekonstruktionsalgorithmus wurde für die Ausführung auf der GPU parallelisiert und optimiert. Lediglich für das Einlesen der Eingabebilder sowie für das Speichern der Tiefenkarten und 3D-Modelle findet eine Datenübertragung zwischen CPU und GPU statt. Eine Einschränkung des Verfahrens besteht darin, dass mögliche Lücken im dreidimensionalen Polygonnetz nicht näher analysiert, sondern einfach geschlossen werden. An dieser Stelle ist eine Weiterentwicklung möglich, indem beispielsweise das Modell in Hinblick auf dessen Symmetrieeigenschaften untersucht und die lückenhaften Bereiche mittels erfolgreich rekonstruierten Modellteilen aufgefüllt werden. Da die Berechnungen zur Rekonstruktion eines 3D-Objektes mithilfe von reziproken Bildpaaren immer noch sehr rechenintensiv und zeitaufwendig sind, besteht auch hier Optimierungsbedarf.

2.7 Ein Verfahren zur Bildkompression

Zur Rekonstruktion eines dreidimensionalen Objektes aus reziproken Bildpaaren, müssen viele einzelne Bilder geladen werden, welche anschließend auf die Grafikkarte übertragen und dort für die Generierung der Tiefenkarten verwendet werden. Die einzeln erstellten Tiefenkarten werden am Ende als Graustufenbilder abgespeichert. Während des Verfahrens müssen also viele Bildinhalte im Grafikspeicher gehalten werden. Um den für die Bilddaten benötigten Speicherbedarf zu reduzieren, ist es daher sinnvoll diese in geeigneter Form zu komprimieren. Werden komprimierte Bildinformationen zur Verarbeitung herangezogen, sollte es mittels eines effizienten Verfahrens möglich sein, die Originaldaten der Bilder in kurzester Zeit wiederherzustellen. Ein solches Verfahren soll in diesem Kapitel vorgestellt werden [GMG10]. Da dessen Anwendung nicht auf die 3D-Rekonstruktion beschränkt ist, sondern u.a. für das Rendering von Texturen in Echtzeit verwendet werden kann, wird der Bezug zum Rekonstruktionsverfahren bei den algorithmischen Erläuterungen nicht weiter berücksichtigt.

Die Kompression von Bildern oder Texturen höherer Auflösung ist in vielen Bereichen der Computergrafik ausschlaggebend. Aufgrund der Programmierbarkeit moderner Grafikkarten ist es möglich, die volle parallele Rechenleistung einer GPU für Kompressions- bzw. Dekompressionsalgorithmen auszuschöpfen, was jedoch gewissen Beschränkungen unterliegt, welche auf die Speicherkapazität einer Grafikkarte zurückzuführen sind. So ist beispielsweise die Auflösung einer Textur bei einer Grafikkarte mit 1.5GB Speicher auf 8192×8192 Pixel beschränkt.

Mit dem für Rastergrafiken entwickelten Dateiformat JPEG2000 [TM01] fand die Wavelet-Kodierung Einzug in die Bildkompression, deren Vorteile in der einfachen Implementierung und in den Adaptionsmöglichkeiten an vorhandene Hardware liegen [WWHL04]. Die verlustbehaftete Kompression wird dabei durch die Quantisierung der Wavelet-Koeffizienten erzielt. Um Speicherplatz zu sparen, können die Wavelet-Koeffizienten in einer Baumstruktur angeordnet und alle Teilbäume, welche die Werte Null enthalten,

entfernt werden. Somit können selbst Texturen höherer Auflösung in den Grafikspeicher transferiert werden, ohne diese in mehrere Teilbilder zerlegen zu müssen.

Auf Basis dieser Erkenntnis wurde eine kompakte Baumdatenstruktur zur effizienten Kompression hochwertiger, zweidimensionaler Bilddaten und ein in Echtzeit, auf der GPU ausführbares Dekompressionsverfahren entwickelt [GMG10]. Der hier vorgestellte Ansatz kann zudem in einfacher Weise für mehrdimensionale bzw. nicht-lineare HDR Daten erweitert werden.

2.7.1 Verwandte Arbeiten

Die S3-Gesellschaft führte 1998 fünf einfache Kompressionsschemata zur Bildkompression ein. Diese basieren auf der Auswertung von 8-Bit RGBA Blöcken und können eine Kompressionsrate von 4:1 bzw. 8:1 erreichen [S3T98]. Zudem wurden diese Techniken von Microsoft für die Entwicklung des DirectX-Frameworks adaptiert. Aufgrund der Tatsache, dass eine Verarbeitung von großen Texturen und die daraus resultierenden Datenmengen, welche unter anderem für das Rendering von Terrain notwendig sind, mit heutigen Grafikkarten nur eingeschränkt oder gar nicht möglich ist, wurde von Tanner et al. [TMJ98] ein *Clipmapping*-Verfahren entwickelt, mit welchem überdimensionale Texturen in mehrere kleinere Einzelbilder bzw. *tiles* aufgeteilt und dadurch in den Grafikspeicher geladen werden können.

Über den JPEG2000-Standard [CSE00, TM01] treten erstmals Wavelets, wie beispielsweise das *LeGall-* oder *Cohen-Daubechies-Feauveau 7/5* Wavelet, im Bereich der Bildkompression in Erscheinung, welche die bis dahin übliche *Diskrete Cosinus Tranformation*, wie sie im regulären JPEG-Format verwendet wird, ersetzen sollte. Die Wavelet basierten Ansätze sind gegenüber anderen Techniken weitaus flexibler in der Handhabung und liefern mit höheren Kompressionsraten auch qualitativ hochwertigere Resultate. Allerdings ist die komplexe Dekomprimierung sehr rechenintensiv, weswegen sich jüngste Forschungsarbeiten auf die Verwendung von modernen Grafikkarten fokusieren, um interaktive Frameraten zu erzielen.

Beers et al. [BAC96] publizierten ein Verfahren, welches auf einer Vektorquantisierung beruht und ein vorberechnetes sogenanntes *Codebook* nutzt, um damit Texturen zu verwalten. Die Größe des *Codebooks* wird dabei durch die Kompressionsstufe festgelegt. Fenny [Fen03] beschreibt einen Weg, wie eine komprimierte Textur lediglich mit einem *Lookup* pro Sample dekomprimiert werden kann. Schneider et al. [SW03b] veröffentlichten ein Kompressionsprinzip für statische und zeit-variable Volumendatensätze. Dieses baut auf einer Vektorquantisierung mit fester Bitrate auf. Für die Kompression wird wiederum ein *Codebook* verwendet, welches eine durch die Bitrate beschränkte Anzahl von generierten Einträgen enthält. Mit dieser Methode kann eine Kompressionsrate von nahezu 20:1 erreicht werden.

Shaprio [Sha01] präsentierte ein Wavelet-Algorithmus mit eingebettetem *Zerotree* (engl. *embedded zerotree wavelet* - EZW). Hierbei werden die aus einer *Diskreten Wavelet Transformation* resultierenden Informationen, wie beispielsweise die Positionen der Wavelet-Koeffizienten, in als *Maps* bezeichnete Abbilder des Originals abgelegt. Im Anschluss werden die signifikanten *Maps* mittels *Zerotree Coding* in einer kompakten Multiskalen Repräsentation gespeicht. Dieser Ansatz bietet eine gute Performanz bei niedriger Komplexität. Nachteilig am EZW ist jedoch, dass sämtliche berechneten Koeffizienten anhand eines bestimmten Schwellwertes klassifiziert und dementsprechend entweder weiterhin berücksichtigt oder komplett vernachlässigt werden. Dies führt dazu, dass einerseits zwar Störsignale bzw. *Noise* in uniformen Regionen entfernt, andererseits verschwommene Artefakte im rekonstruierten Bild generiert werden. DiVerdi et al. [DCH05a] bieten eine Implementierungsmöglichkeit des EZW an, mit welcher eine Dekodierung auf der GPU vollzogen werden kann. Die per *Haar*-Wavelet kodierten Koeffizienten werden innerhalb eines Baumes angeordnet, welcher über einen *Zero*-Knoten verfügt, auf welchen alle Baumknoten verweisen, deren Koeffizienten sich dem Wert Null annähern. Diese Vorgehensweise führt zu guten Kompressionsraten, ist jedoch problematisch bei der Verarbeitung von verrauschten Bilddaten, da Koeffizienten nahe Null einfach ignoriert werden und dies in wahrnehmbaren Abweichungen resultiert.

2.7.2 Die Baumbasierte Bildkompression

Da das menschliche Sehempfinden sensibler auf Helligkeitsveränderungen in Bildern reagiert, als auf deren Farbunterschiede, wird vor der eigentlichen Kompression jedes Eingabebild einer Gammakorrektur unterzogen. Dabei werden sämtliche *RGB*-Werte in den YP_bP_r-Farbraum überführt, wobei Y für die Helligkeits- bzw. Luminanzinformation steht und P_b bzw. P_r die Werte zur Farbdifferenz bzw. Chrominanz repräsentieren. Nach der Farbraumkonvertierung kann die Wavelet-Transformation (siehe Kapitel 2.1.5) durchgeführt werden. Für die Bildkompression ist dabei die Wahl der Waveletbasis maßgeblich. Vor allem sollte auf die Trägerbreite, dem lokalen Verhalten im Zeitbereich, geachtet werden. Eine höhere Trägerbreite führt zu besseren Kompressionsraten, steigert jedoch den Rechenaufwand. Um die Vor- und Nachteile einer Waveletbasis bzw. deren Auswirkungen auf die Bildkompression zu verdeutlichen, wurde bei der Umsetzung des Verfahrens neben dem *Haar*-Wavelet, das biorthogonale *LeGall*- und das quadratische *B-Spline* Wavelet implementiert.

Durch das *Haar*-Wavelet, welches eine kompakte Trägerbreite von 1 aufweist, kann aufgrund seiner Einfachheit die Performanz in der Dekodierungsphase optimiert werden. Allerdings ist dieses weder kontinuierlich noch differenzierbar, was sichtbare Blockartefakte im komprimierten Bild hervorrufen wird.

Das biorthogonale *LeGall*-Wavelet, auch als Cohen-Daubechies-Feauveau 5/3 Wavelet bekannt, wird für die verlustfreie Kompression im JPEG2000-Standard [CSE00, TM01] verwendet und ist kontinuierlich aber nicht differenzierbar. Im Vergleich zum *Haar*-Wavelet werden die lokalen Veränderungen im Frequenzbereich geglättet, was sich in bessere Kompressionsergebnisse auswirken sollte.

Das quadratische *B-Spline* Wavelet [SB04] ist sowohl kontinuierlich als auch differenzierbar, sodass unter Verwendung dieser Wavelet-Basis die besten Kompressionsresultate erwartet werden. Da es über dieselbe Trägerbreite wie das *LeGall*-Wavelet verfügt, sollte das Leistungsverhalten bei der Dekompression der Daten äquivalent sein.

Bei gängigen Bildkompressionsverfahren wie JPEG2000, wird nach der Wavelet-Transformation eine Entropiekodierung der quantisierten Waveletkoeffizienten durchgeführt. Da eine Entropie-basierte Kodierung der Koeffizienten für eine Echtzeit-Dekompression auf der GPU ungeeignet ist, werden im entwickelten Verfahren die Waveletkoeffizienten zunächst in einer baumartigen Datenstruktur angeordnet und im Anschluss solange Redundanzen entfernt bzw. Knoten mit identischen oder ähnlichen Werten iterativ zusammengefasst, bis die gewünschte Kompressionsrate erreicht wird. Durch diese Vorgehensweise ergibt sich letztendlich ein allgemeiner gerichteter Graph, welcher in einer 3D-Textur gespeichert werden kann.

2.7.2.1 Die Baumstruktur

Bei der dyadischen Dekomposition wird ein Bild pro Auflösungsstufe in vier gleichgroße Blöcke untergliedert (siehe Abbildung 2.32 links). Jedem Pixel (bzw. LL-Koeffizienten) einer aktuellen Detailstufe können die entsprechenden LH-, HL- und HH-Koeffizienten zugeordnet werden. Um einen Pixel in die nächst höhere Auflösungsstufe überführen zu können, muss dieser aufgrund der Bildunterteilung mit jeweils vier LH-, HL- und HH-Koeffizienten multipliziert werden. Basierend auf diesen Abhängigkeiten kann eine Baumstruktur konstruiert werden, in welcher zu einem Pixel der aktuellen Auflösung die zugehörigen Waveletkoeffizienten und die vier Pointer zum nächst höheren Level, zu einem Knoten zusammengefasst werden. Der LL-Koeffizient für die niedrigste Auflösung sollte dabei separat von der Baumstruktur gespeichert werden. Mit dieser Vorgehensweise ist es möglich einen einzelnen Bildpunkt zu rekonstruieren, indem ausgehend vom Wurzelelement der komplette Baum durchlaufen und entsprechende Koeffizienten eingesammelt werden.

In Hinblick auf den Speicherverbrauch ist diese Baumstruktur jedoch verhältnismäßig ineffizient. Wenn 24 Bits als Kerngröße veranschlagt werden, stehen den 9 Bytes, welche für die LH-, HL- und HH-Koeffizienten pro Pixel benötigt werden, 12 Bytes für die relevanten Pointer gegenüber. Somit muss mehr Speicher für die Verzweigungen innerhalb der Baumstruktur aufgebracht werden, als für die eigentlichen Bilddaten erforderlich ist. Um den Pointer-Overhead zu verringern, werden in der entwickelten Baumstruktur die

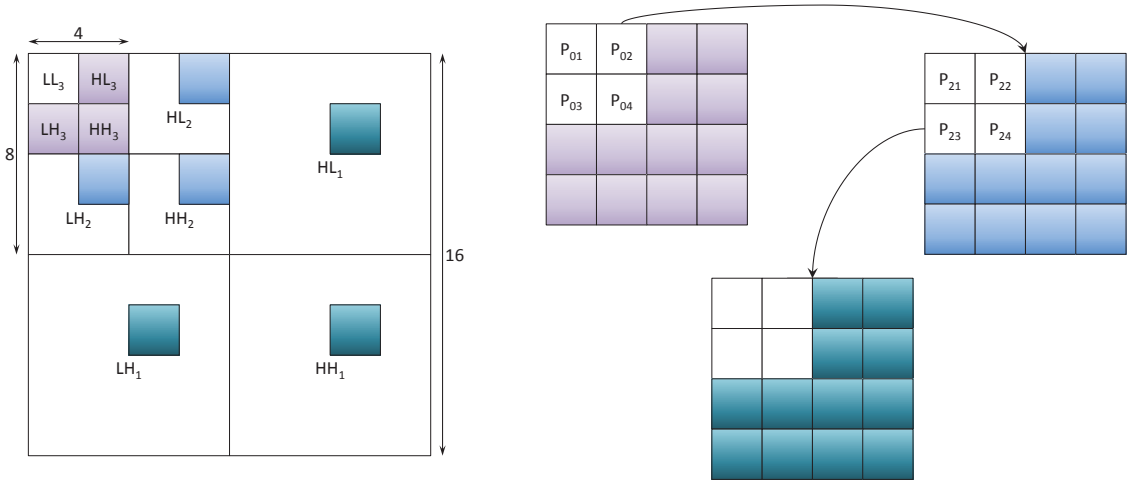


Abbildung 2.32: Schematische Darstellung der dyadischen Zerlegung (links) und der daraus abgeleiteten Datenstruktur (rechts). Mithilfe der Pointer können die für die Rekonstruktion eines Bildpunktes relevanten Waveletkoeffizienten erreicht werden. Die farbliche Unterscheidung soll die Lagebeziehung der Koeffizienten in der Datenstruktur zur korrespondierenden Position im transformierten Bild verdeutlichen.

Koeffizienten für einen 2×2 großen Pixelblock (also für ingesamt vier Bildpunkte) zu einem Knoten gruppiert. Demnach beinhaltet jeder Knoten 12 quantifizierte Koeffizienten (jeweils 4 LH-, HL- und HH-Werte) und 4 Pointer (siehe Abbildung 2.32 rechts). Dadurch beschränkt sich der Overhead pro Datenblock (36 Bytes) auf 12 Bytes, was einer Ersparnis von 75% entspricht. Aufgrund der 2×2 Blockeinteilung wird die niedrigste Auflösungsstufe allerdings durch 4 LL-Koeffizienten repräsentiert, welche immer noch getrennt von der Baumstruktur gespeichert werden müssen.

2.7.2.2 Kompression der Baumstruktur

Nachdem die baumartige Datenstruktur aufgebaut wurde, können redundante Informationen aus dem Baum iterativ entfernt werden. Während dieses Vorgangs sollten jedoch nicht ausschließlich die in einem Knoten enthaltenen Koeffizienten auf Ähnlichkeit geprüft, sondern auch die durch die Pointer erreichbaren Teilbäume betrachtet werden. Nur wenn neben den Koeffizienten auch die Kindknoten übereinstimmen oder entsprechend eines Fehlermaßes als ähnlich eingestuft wurden, können Knoten miteinander kombiniert werden. Mit dieser Vorgehensweise lässt sich ein allgemeiner, gerichteter Graph mit einem Wurzelelement konstruieren (siehe Abbildung 2.33).

Da während der Verschmelzung zweier Knoten Approximierungsfehler auftreten und sich pro Iteration fortsetzen können, ist die Wahl der Knoten, welche zusammengefasst werden sollen, sowie die Reihenfolge der Kollapsoperationen entscheidend für die Qualität der in der Dekompressionsphase erzielten Resultate. Daher wird im entwickelten Verfahren eine Priority-Queue verwendet, durch welche die einzelnen Operationen in einer optimalen

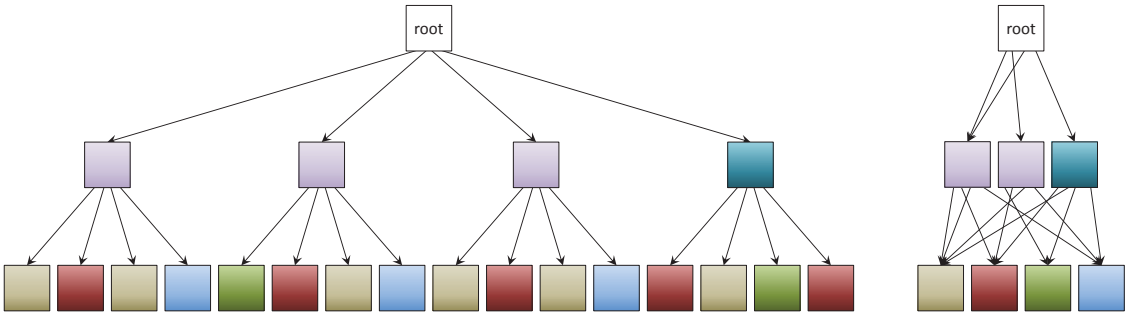


Abbildung 2.33: Die gerichtete Graphen-Datenstruktur unkomprimiert (links) und komprimiert (rechts). Um die verschiedenen, in einem Knoten gespeicherten Koeffizienten differenzieren zu können, wurden diese farblich hervorgehoben. Knoten mit identischen Koeffizienten bzw. Kindknoten können zusammengefasst werden.

Reihenfolge auf den gerichteten Graphen ausgeführt werden. Für die Sortierung der Operationen wurde ein Fehlermaß definiert, welches sich proportional zur Summe der quadratischen Abweichung (engl. *sum of squared differences* - SSD) über alle Knoten, die in einer Operation involviert sind, verhält.

Seien i und j zwei mögliche Kandidaten, welche zu einem Knoten k zusammengefasst werden sollen und w_i bzw. w_j Werte für eine adäquate Gewichtung des zugehörigen Fehlers ε_k, so lässt sich dieser aus den beiden Fehlerwerten ε_i bzw. ε_j und der gewichteten Summe der quadratischen Differenzen $d^2(i,j)$ folglich bestimmen:

$$\varepsilon_k = \varepsilon_i + \varepsilon_j + d^2(i,j)\frac{w_i w_j}{w_i + w_j}, \tag{2.71}$$

wobei die Werte für ε_i bzw. ε_j entfallen, falls es sich bei den Knoten i bzw. j um Blätter innerhalb der Baumstruktur handelt und $d^2(i,j)$ mittels der in i bzw. j enthaltenen Koeffizienten bestimmt werden kann. Das neue Gewicht w_k für den Knoten k ergibt sich aus Summe von w_i und w_j. Um die optimale Reihenfolge der auszuführenden Operationen zu gewährleisten, werden diese anhand des berechneten Fehlers in die Priority-Queue eingefügt. Dabei ist die Priorität einer Operation um so höher, desto niedriger der entsprechende Fehler ausfällt. Mit anderen Worten kann ein Knoten i nur mit einem anderen kollabiert werden, wenn der jeweilige Fehler minimal ist. Demzufolge muss also der optimale Kandidat für i gefunden und diese Paarung in der Priority-Queue abgelegt werden. Dieses Problem kann auf die *nearest neighbor search* in mehrdimensionalen Raum zurückgeführt werden (für eine genauere Beschreibung sei auf das Ende dieses Abschnitts verwiesen). Die einzige Ausnahme bilden dabei die Knoten, deren Kindknoten nicht übereinstimmen. In diesem Fall wird im entwickelten Verfahren die Distanz zwischen zwei Knoten, welche jeweils durch einen 36-dimensionalen Vektor (für jeden der drei Farbraumkomponenten Y, P_b und P_r 12 Koeffizienten) repräsentiert werden, auf unendlich gesetzt, sodass deren Verschmelzung vermieden wird.

Sobald eine Operation ausgeführt wurde, werden einige in der Queue enthaltenen Einträge ungültig und müssen neu berechnet werden. Dies ist beispielweise der Fall, wenn zwei Knoten zusammengeführt wurden und der resultierende neue Knoten sich als vorteilhafterer Kandidat für eine andere in der Queue befindliche Paarung herausstellt. In der Annahme, dass sich für den neuen Knoten eine höhere SSD ergibt und aufgrund dessen dieser nur für die Folgeoperationen infrage kommt, kann die Aktualisierung dieser Daten solange hinausgezögert werden, bis die aktuelle Operation aus der Queue entfernt wurde. Lediglich für den neu entstandenen Knoten und für die Knotenpaare, bei welchen einer der kollabierten Knoten in den untergeordneten Teilbäumen zu finden ist, sollte eine sofortige Suche nach dem *nearest neighbor* angestrebt werden. Die Betrachtung der entsprechenden Knotenpaare ist notwendig, da sich aufgrund der höheren SSD des neuen Knotens ein anderer Baumknoten als geeigneter für die Verschmelzung herausstellen könnte.

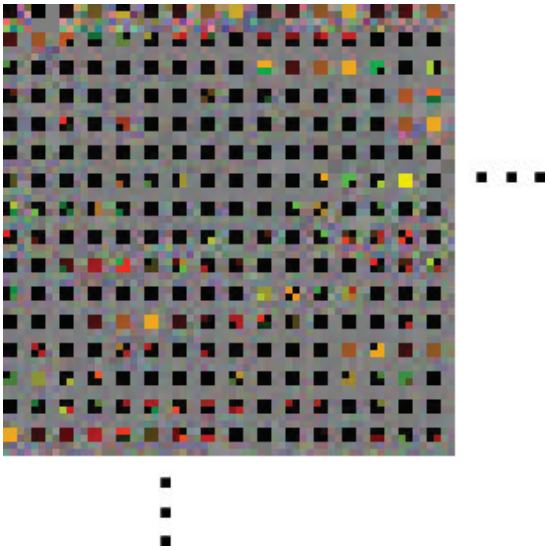


Abbildung 2.34: Ein Teil der komprimierten Bilddaten, welcher in einer 3D-Textur gespeichert wurde.

Natürlich kann bei der Anordnung der Operationen (neben dem definierten Fehlermaß) auch der Umstand ausgenutzt werden, dass innere Baumknoten nur dann kollabiert werden können, wenn deren Blatt- bzw. Kindknoten bereits zusammengeführt wurden. Andernfalls wäre die anfänglich festgelegte Bedingung der identischen Teilbäume nicht gegeben. Die komprimierte Datenstruktur kann im Anschluss in einer 3D-Textur gespeichert werden (siehe Abbildung 2.34).

Um die geschilderte Vorgehensweise zu optimieren bzw. eine Reduktion der durchzuführenden Operationen herbeizuführen, werden vor dem Aufbau der Priority-Queue sämtliche Blattknoten, deren enthaltene Koeffizienten zu dem Wert Null quantisiert wurden, zu einer *zero node* zusammengefasst. Diese Optimierungsstrategie wird als *zerotree coding* bezeichnet und wurde von DiVerdi, Candussi und Höllerer im Jahre 2005 entwickelt [DCH05b]. Da für das *zerotree coding* keine *nearst neighbor search* oder Priority-Queue notwendig ist, können Bestandteile der Baumstruktur, welche keinerlei relevante Informationen beinhalten, einfach und effektiv entfernt werden. Im Gegensatz zu [DCH05b], werden im hier beschriebenen Ansatz Knoten, mit Koeffizienten nahe dem Wert Null, jedoch nicht einfach mit der *zero node* kollabiert, sondern erst mit einem Knoten kombiniert, sobald die zugehörige SSD klein genug ist. Da durch diesen zusätzlichen Schritt die Anzahl der in der Datenstruktur befindlichen Knoten reduziert wird, lässt sich der Aufwand für die *nearest neighbor search* und die maximale Anzahl der zu bewältigenden Kollapsoperationen vermindern, was sich widerum positiv auf die Lautzeiten auswirkt. Dies ist im Speziellen

für Bilder von Bedeutung, deren Dimension vor der Wavelet-Transformation angepasst bzw. aufgefüllt werden muss.

02. Kompression der Baumstruktur **Nearest Neighbor Search**

In den vorangegangen Erläuterungen wurde oft von der *nearest neighbor search* gesprochen, nähere Details zur Vorgehensweise wurden jedoch verschwiegen. Dies soll nun nachgeholt werden. Wie bereits erwähnt, wird ein Knoten der Baumstruktur durch einen 36-dimensionalen Vektor repräsentiert, welcher pro Farbkanal 12 Wavelet-Koeffizienten beinhaltet. Zu jedem dieser Vektoren soll ein möglicher Kandidat zum Zwecke des Kollabierens gefunden und der zugehörige Fehler minimiert werden. Es wurde also eine effiziente Methode zur Suche des geeignetesten Nachbarn in einem 36-dimensionalen Raum benötigt. Da jede Operation das Hinzufügen eines und das Entfernen zweier Vektoren aus der Datenmenge impliziert, kann eine räumliche Datenstruktur wie beispielsweise der *R-Tree* [Gut84] zu Beschleunigungszwecken nicht eingesetzt werden. Aus diesem Grund beschränkt sich die *nearest neighbor search* auf die Definition einer linearen Ordnung, basierend auf der Sortierung von Schlüsselwerten. In diesem Zusammenhang kann die Tatsache ausgenutzt werden, dass sich viele der Koeffizienten-Vektoren, mit einer mehr oder weniger gaußschen Verteilung, um ein räumliches Zentrum herum anordnen. Somit kann die Distanz zum Zentrum als Auswahlkriterium verwendet werden, indem nach einem Knoten bzw. Vektor mit ähnlichem Abstand gesucht wird. Sobald ein entsprechender Kandidat gefunden wurde, kann die Suche eingestellt werden, da Vektoren mit einer größeren oder viel kleineren Distanz zum Zentrum, zu größeren Abweichungen zwischen den jeweils betrachteten Knoten führen und daraus höhere Fehlerwerte resultieren würden.

2.7.3 Parallele Dekompression

Um die komprimierte Graphen-Datenstruktur in einer 3D-Textur ablegen zu können, müssen einige durch die Grafik-Hardware bedingte Vorgaben beachtet werden. So ist die Größe einer dreidimensionalen Textur auf $256 \times 256 \times 2^n$, mit $0 \leq n \leq 8$, festgelegt und für jeden einzelnen Pixel stehen 24 Bits im regulären RGB-Format zur Verfügung. Für Letzteres müssen die quantisierten Koeffizienten, welche sich im Wertebereich $[0, 1]$ befinden, auf das Intervall $[0, 255]$ skaliert und vom YP_bP_r-Format zurück in den RGB-Farbraum transferiert werden.

Wie in Abbildung 2.34 zu erkennen, werden die Knoten der Baumstruktur in Blöcken von 4×4 Pixel kodiert. Die vier in der oberen linken Ecke eines Blockes befindlichen Pixel entsprechen den Pointern, welche auf die zugehörigen Kindknoten verweisen. Die Referenz dieser Zeiger wird an dieser Stelle als Farbwert interpretiert und kann somit direkt in Textur-Koordinaten umgewandelt werden.

Für eine parallele Dekompression auf der GPU, muss der gegenwärtig betrachtete Bildpunkt mit denen durch die verwendete Waveletfunktion vorgegebenen Waveletkoeffizienten ausgewertet und multipliziert werden. Die Anzahl der Waveletkoeffizienten ist dabei abhän-

gig von der Breite des Mutter-Wavelets. Bei der Umsetzung dieses Verfahrens wurde neben dem *Haar*-, das *LeGall*- sowie das quadratische *B-Spline* Wavelet betrachtet. Ersteres verfügt über eine Trägerbreite von 1, wohingegen die beiden anderen eine Breite von 3 (pro Dimension) aufweisen. Somit müssen insgesamt 3 Koeffizienten für das $Haar$-Wavelet und 27 für die beiden anderen Wavelets pro Auflösungsstufe für die Rekonstruktion eines Bildpunktes berücksichtigt werden. Die jeweiligen Werte ergeben sich aus der Breite pro Dimension und den entsprechend einzubeziehenden LH-, HL- und HH-Koeffizienten (also $3 \times Breite \times Breite$). Um die passenden Koeffizienten aus der Baumstruktur extrahieren zu können, müssen ein (*Haar*) bzw. vier (*LeGall, B-Spline*) Knoten inspiziert werden. Dies führt letztlich zu 4 bzw. 31 notwendigen Textur-Lookups pro Level (Anzahl der Koeffizienten plus relevante Pointer). Da es sich in der höchsten Auflösungsstufe bei den Knoten um Blätter handelt und somit keinerlei Pointer zu weiteren Kindknoten führen, ergibt sich eine Gesamtanzahl von *4l-1* Lookups für die Verwendung des *Haar*-Wavelets bzw. *31l-4* für das *LeGall*- und *B-Spline* Wavelet, wobei l der Anzahl der in der Baumstruktur enthaltenen Ebenen entspricht.

Obgleich die Anzahl der Lookups für die komplexeren Wavelets ziemlich hoch erscheint, ist im Allgemeinen beim Einsatz des *Haar*-Wavelets keine bilineare Interpolation möglich. Dies führt unter anderem zu einer geringeren Bildqualität und einer stärkeren Ausbildung von Artefakten, sobald Formatänderungen am Bild (Zoom) vorgenommen werden. Um die bilineare Interpolation zu ermöglichen, würde sich die Anzahl der notwendigen Lookups auf *16l-4* vervierfachen, was in Hinblick auf die beiden anderen Wavelets immer noch ungefähr der Hälfte entspricht. Da sich jedoch die komplexeren Waveletfunktionen durch feinere Signalübergänge auszeichnen und aufgrund dessen bessere Ergebnisse bei gleicher Kompressionsrate erzielt werden können, erscheint die höhere Anzahl von Textur-Lookups durchaus akzeptabel.

2.7.4 Ergebnisse

Um den vorgestellten Algorithmus evaluieren und mit existierenden Kompressionsverfahren vergleichen zu können, wurden Bilder aus der Sammlung der *Image Compression Benchmark* [Gar] verwendet (siehe Abbildung 2.36 und 2.37). Mithilfe dieser Datenbank ist es möglich, die Qualität einer Kompressionsmethode zu bewerten und entsprechend einzustufen.

Für die Umsetzung der eigenen Kompressionstechnik wurden neben dem *Haar*-Wavelet, das *LeGall*- und das *B-Spline*-Wavelet implementiert. Die damit im Zusammenhang stehenden qualitativen Unterschiede zwischen den jeweiligen Kompressionsergebnissen werden in Abbildung 2.35 verdeutlicht. Bei der Verwendung des *Haar*-Wavelets werden in den Resultaten signifikante Blockartefakte sichtbar, welche mit steigender Kompressionsrate zunehmen. Diese Blockartefakte können mithilfe des *LeGall*- bzw. *B-Spline*-Wavelets vermieden werden. Da es sich bei der Skalierungsfunktion des *LeGall*-Wavelets jedoch

Abbildung 2.35: Vergleich zwischen den verwendeten Wavelet-Transformationen (von links nach rechts): *Haar*-, *LeGall*- und *B-Spline*-Wavelet.

um einen linearen Filter handelt, kann dies zur Generierung sternenförmiger Artefakte innerhalb eines komprimierten Bildes führen. Zudem wirken einige Bildbereiche verschwommen. Der Vorteil des quadratischen *B-Spline* Wavelets besteht darin, dass die im Bild enthaltenen Details klarer reproduziert und aufgrund der bi-quadratischen Interpolation bei einer im Nachhinein durchgeführten Vergrößerung der Bilder bessere Resultate erzielt werden. Das *Spitzen-Signal-Rausch-Verhältnis* (engl. *peak signal-to-noise ratio* - PSNR), welches einen Maßstab zur Messung des Qualitätsverlusts zwischen Original und komprimierten Bild wiederspiegelt, ist bei den betrachteten Waveletfamilien ungefähr gleich, wobei die Werte für das *Haar*-Wavelet (41.7 dB) im Gegensatz zum *LeGall*- (45.4 dB) bzw. *B-Spline*-Wavelet (42.9 dB) niedriger ausfallen.

Abbildung 2.36: Ergebnisse zur Bildkompression mithilfe von *Embedded Zero Tree Coding* (links) und dem vorgestellten Verfahren (rechts).

In Abbildung 2.36 wird der Unterschied zwischen dem *Embedded Zerotree Coding* [Sha01] und der im Rahmen dieser Arbeit entwickelten Kompressionsmethode veranschaulicht. Das Originalbild wurde in beiden Fällen bei einer Kompressionsrate von 23:1 komprimiert, wobei die PSNR für den Ansatz von Shapiro einen Wert von 35.4 dB lieferte und für den vorgestellten Algorithmus einen Wert von

37.6 dB. Ein Nachteil des *Embedded Zero Coding* liegt darin, dass alle Koeffizienten unterhalb einer bestimmten Fehlerschwelle einfach vernachlässigt werden. Zwar können dadurch Störsignale in uniformen Bildregionen reduziert, hochfrequente Bilddaten jedoch nicht komprimiert werden. Um eine gewünschte Kompressionsrate bei Bildern zu erzielen, in welchen hohe Frequenzanteile enthalten sind, muss der festgelegte Schwellwert angehoben werden. Unter Berücksichtigung des zuvor genannten Nachteils, führt dies allerdings zu einer Qualitätsminderung bei den Kompressionsergebnissen. Im Gegensatz zum *Embedded Zerotree Coding* können mit dem entwickelten Clusteringansatz Ähnlichkeiten in hochfrequenten Bildregionen aufgedeckt und entsprechende Knoten mit der *zero node* kollabiert werden, was die visuelle Qualität komprimierter Bilddaten verbessert. Des Weiteren können mit dem angegebenen Kompressionsverfahren mögliche *Blur*-Effekte verringert und die Details verschiedener Bildregionen deutlicher voneinander abgegrenzt werden.

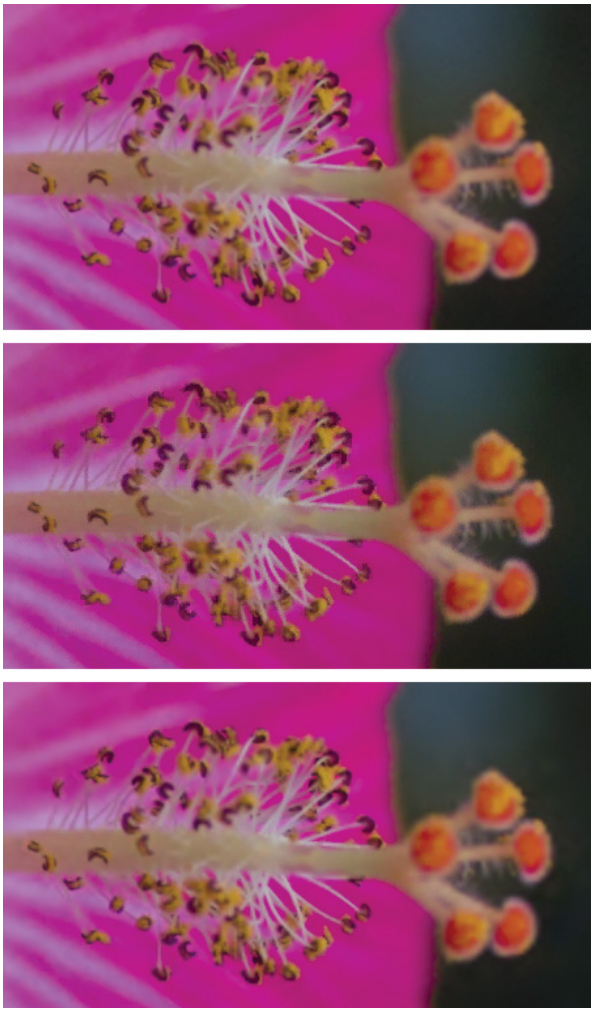


Abbildung 2.37: Vergleich zwischen S3TC (Mitte) und der eigenen Kompressionsmethode, bei gleicher Kompressionsrate (oben) und gleicher Qualität (unten).

Abbildung 2.37 soll den Vergleich mit S3TC [S3T98] veranschaulichen. Für eine hinreichende Analyse wurden die Bilddaten zunächst bei einer Kompressionsrate von 6:1 (8:1 bei Bilder im RGBA-Format) sowohl mit dem von S3TC entwickelten Verfahren als auch nach der vorgestellten Vorgehensweise kodiert. Anschließend wurde versucht, die durch die S3TC-Methode erreichte Qualität mittels der eigenen Kompressionstechnik nachzubilden. Die zwei oberen in Abbildung 2.37 enthaltenen Bilder wurden bei gleicher Kompressionsrate generiert, wobei sich im Bezug auf S3TC eine PSNR von 55.1 dB und für den eigenen Kompressionsansatz eine noch bessere Qualtität (60.7 dB) erzielen ließ. Gegenüber der S3TC-Methode konnte zusätzlich der für die komprimierten Bilddaten benötigte Speicherbedarf gesenkt werden (bzgl. des Beispiels von 1.6 MB auf 1.5 MB). Wird die Kompressionsrate vervierfacht (26:1), ist die visuelle Qualität, welche durch den eigenen Ansatz erreicht wird (57.1 dB), immer noch der von S3TC überlegen bzw. zu dieser äquivalent.

Selbst bei einer 3000×3000 Pixel großen Luftaufnahme (siehe Abbildung 2.38) kann bei einer sehr hohen Kompressionsrate von 34:1 eine ausreichende Qualität (36.4 dB) gewährleistet werden, bei welcher nach wie vor alle dominierenden Details erkennbar sind.

Abbildung 2.38: Hochgradige Kompression (34:1) einer Luftaufnahme (links) inklusive einer Vergrößerung des darin markierten Bereiches (rechts).

Die Dekodierung der Bilddaten wurde in Echtzeit auf einer NVIDIA GeForce GTX 295 durchgeführt. Bei einer 4096 × 4096 großen Textur konnten mithilfe des *Haar*-Wavelets und der *Nearest Neighbor Search* ungefähr 400 MPixel pro Sekunde (ca. 100 MPixel unter Verwendung von bilinearen Filtern) verarbeitet werden. Für die mithilfe des *LeGall-* bzw. des *B-Spline*-Wavelets vollzogene Bildkompression kann eine Performanz von etwa 55 MPixel pro Sekunde erwartet werden. Bei kleineren oder größeren Bildformaten sinkt bzw. erhöht sich die Laufzeit linear mit der Anzahl der bei der Wavelet-Dekomposition betrachteten Level, wobei mit der entwickelten Methode selbst bei einer 16k × 16k Textur eine Leistung von 46 MPixel pro Sekunde erreicht werden kann.

2.7.5 Zusammenfassung

In den letzten Abschnitten wurde eine effektive Methode zur Bildkompression präsentiert, welche auf der Basis einer Wavelet-Transformation arbeitet und auf große bzw. hochauflösende Texturen angewendet werden kann. Mithilfe der innovativen, baumartigen Datenstruktur können bei einer hohen visuellen Bildqualität im Gegensatz zu vergleichbaren Verfahren höhere Kompressionsraten erzielt werden, besonders gegenüber dem einfachen *zerotree coding*. Die Dekompression der Bilddaten wurde als Pixel-Shader implementiert, welcher auf der aktuellen Grafikhardware in Echtzeit ausgeführt werden kann.

Da für die Kompression eines 67 MPixel großen Bildes (8192 × 8192 Pixel) auf einer NVIDIA GeForce GTX 295 zwar eine im Vergleich zu bekannten Techniken geringere, aber dennoch hohe Kompressionszeit von ungefähr 30 Minuten benötigt wird, sollte das Verfahren zukünftig weiter optimiert werden, um die Kompressionszeiten zu verbessern. Des Weiteren wäre eine Erweiterung dieser Methode zur Verarbeitung von *high dynamic range* (HDR) und multidimensionalen Daten realisierbar.

KAPITEL 3

Verarbeitung von dreidimensionalen Polygonnetzen

In der Animationstechnik und der Spieleprogrammierung bilden digitale, dreidimensionale Objekte die Basis für die Entwicklung virtueller Welten. Die einzelnen 3D-Modelle werden dabei in der Regel nicht direkt an einem Rechner generiert. Vielmehr werden zunächst mithilfe von Materialien wie z.B. Ton oder Gips Prototypen nach realen Vorbildern modelliert (*sculpting*) oder es wird nach bereits vorhandenen Gegenständen bzw. Raumsituationen gesucht, die im Anschluss fotografiert oder eingescannt werden. Auf dem Computer werden aus den resultierenden Daten dreidimensionale Polygonnetze erzeugt, welche schließlich von Hand nachbearbeitet, deformiert oder mit zusätzlichen Features erweitert werden können.

Die virtuellen Welten werden meistens so konzipiert, dass diese für den Endverbraucher den Eindruck einer realitätsnahen Umgebung widerspiegeln, um so letztlich eine gewisse Grundstimmung zu erzeugen oder gar gezielt Emotionen hervorrufen zu können. Um in diesem Zusammenhang dem wachsenden Realitätsanspruch Rechnung zu tragen, werden die 3D-Modelle heutzutage mit sehr komplexen Strukturen und vielen kleineren Details versehen. Dies führt unweigerlich zu enorm großen Datenmengen, die gespeichert und verarbeitet werden müssen. Für eine Reduktion des Speicherbedarfs und des Rechenaufwands werden Algorithmen benötigt, mit deren Hilfe 3D-Modelle unter Einsatz einer geeigneten Datenstruktur effizient und ohne Informationsverlust simplifiziert und im Nachhinein korrekt dargestellt werden können. Die grundlegende Idee der Verfahren zur Simplifizierung eines Modells beruht dabei üblicherweise auf dem Kontrahieren bzw. Kollabieren von Kanten, welche durch ein Vertexpaar definiert sind (engl. *Vertex Pair Contraction*). Während einer Simplifizierungssequenz sollte drauf geachtet werden, dass der geometrische Fehler, der durch die *collapse*-Operationen ausgelöst wird, minimal bzw. unterhalb einer akzeptablen Fehlerschranke bleibt.

Im Bereich der Spieleentwicklung ist es zudem von höchster Relevanz, dass die Daten in Echtzeit verarbeitet werden, um lästige Ladezeiten während des Spielablaufs zu vermeiden und damit dem Spieler das optimale Spielerlebnis in der virtuellen Welt ermöglichen zu können. Dies führt oftmals dazu, dass zu den verwendeten 3D-Modellen mehrere simplifizierte

Versionen, die lediglich einen Bruchteil der Originaldaten enthalten, generiert werden und in Verbindung mit Sichtbarkeitsalgorithmen für den Aufbau einer augpunktabhängigen Szenenansicht Verwendung finden.

Neben einer schnellen und effizienten Datenverarbeitung ist zusätzlich die Möglichkeit zur Verformung dreidimensionaler Polygonnetze wünschenswert. Algorithmen, welche über diese Option verfügen, sind in der Lage Änderungen an der Objektgeometrie sowie an deren Topologie vorzunehmen, wodurch es möglich wird komplexe Bewegungsabläufe zu modellieren oder mit geringem Aufwand komplett neue Modelle zu kreieren. Werden derartige Editiermodi mit den Simplifizierungsverfahren kombiniert, müssen die Modifikationen nicht an einem Originalmodell vollzogen werden, wo selbst kleinste Änderungen mit einen erheblichen Bearbeitungsaufwand einhergehen, da beispielsweise einzelne Punkte einer Objektoberfläche manuell verschoben, hinzugefügt oder entfernt werden müssen. Wird hingegen ein Modell vor der angedachten Manipulation auf ein grobes Detaillevel reduziert, können marginale Veränderungen einen großen Einfluss auf das Original ausüben, wenn diese entsprechend propagiert werden.

In den folgenden Kapiteln sollen neben den Verfahren zur Simplifizierung größerer Datenmodelle, Strategien für die Parallelisierung dieser Techniken vorgestellt und mögliche Erweiterungen diskutiert werden, mit deren Hilfe die Modellierung dreidimensionaler Polygonnetze, auf Basis der parallelen Datenstrukturen, verwirklicht werden kann. Vorab sollen jedoch relevante Grundlagen sowie die Herangehensweisen verwandter Forschungsarbeiten beschrieben werden.

3.1 Grundlagen

Bei der Verarbeitung dreidimensionaler Modelle spielen viele verschiedene Faktoren eine Rolle. Da diese jedoch immer im Zusammenhang mit dem jeweiligen Anwendungsfall analysiert und entsprechend angepasst werden müssen, werden in diesem Abschnitt lediglich die grundlegenden Begrifflichkeiten eingeführt ohne dabei näher auf deren algorithmische Abläufe oder Umsetzungsmöglichkeiten einzugehen. Detaillierte Erläuterungen werden an entsprechender Stelle nachgeholt.

3.1.1 Level-of-Detail (LOD)

In der Computergrafik werden komplexe Modelle zur Generierung virtueller Welten verwendet. Die Komplexität dieser Modelle beruht auf dem stetig wachsenden Realitätsanspruch, aufgrund dessen immer feinere Details in die Poloygonnetze der 3D-Modelle integriert werden. Dies führt unweigerlich zu großen Datenmengen und zu einer hohen Anzahl von im Polygonnetz (auch als *Mesh* bezeichnet) enthaltenen Dreiecken, die zur Darstellung einer virtuellen Welt gerendert werden müssen. Um den Aufwand für das Rendering zu minimieren bzw. eine Reduktion der zu zeichnenden Dreiecke herbeizuführen, werden soge-

nannte *Level-of-Detail* (LOD) Verfahren eingesetzt. Hierbei wird der Umstand ausgenutzt, dass mit zunehmendem Abstand zum Objekt, die dort enthaltenen feineren Strukturen vom Betrachter nur noch teilweise oder gar nicht wahrgenommen werden. Somit können beispielsweise Dreiecke, deren Ausmaße aus der Sicht des Betrachters die Größe eines Pixels unterschreiten, entweder komplett ignoriert oder direkt als Punkt dargestellt werden. Zu diesem Zweck kann der *Screenspace Error* ermittelt werden, welcher den Grad der Abweichung zwischen der metrischen Position eines Bildpunktes und seiner Zeichenposition im Pixelformat widerspiegelt.

Bei den Algorithmen zur Generierung von LODs wird zwischen *statischem* und *dynamischem Level-of-Detail* unterschieden. Während beim *statischen* LOD die 3D-Modelle entsprechend der Entfernung zum Betrachter durch in einem Vorverarbeitungsschritt erzeugte, vereinfachte Versionen ihrer selbst ersetzt werden (siehe Abbildung 3.1), wird bei *dynamischen* LODs die Approximation eines Modells zur Programmlaufzeit ermittelt. Letztere erfolgt unter Berücksichtigung von Sichtbarkeitskriterien und muss aktualisiert werden, sobald sich die Position des Betrachters oder der Blickwinkel zur Szene hin verändert.

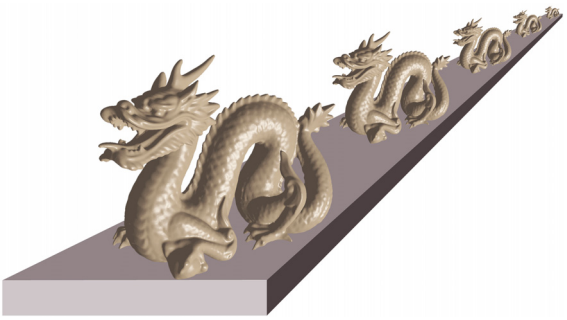


Abbildung 3.1: Beispiel für *statisches* LOD. Mit zunehmender Entfernung kann eine grobere Version des Modells verwendet werden.

Der Vorteil von *statischen* LODs liegt in der einfachen Berechnung der einzelnen Modellversionen. Da jedoch nur eine bestimmte Anzahl von Ansichten generiert wird und kein fließender Übergang zwischen zwei Detailstufen besteht, können störende *Popping-Artefakte* auftreten, sobald eine Detailstufe durch die nächstkleinere oder nächstgrößere ausgetauscht wird. Ein sprunghafter Wechsel zwischen den Modellansichten kann mithilfe eines *dynamischen* Verfahrens verhindert werden. Der erhöhte Rechenaufwand wirkt sich hingegen nachteilig auf den jeweiligen Anwendungsfall aus.

3.1.2 Die Grundoperationen *collapse* und *split*

Die Simplifizierung dreidimensionaler Modelle basiert auf der Ausführung von sogenannten *collapse*-Operationen, die eine Kante zu einem Punkt kollabieren lassen. Die Umkehrung dieser Operationen wird durch den zugehörigen *Vertex Split* beschrieben, bei welchem die ursprünglichen Strukturen wiederhergestellt werden. In Abbildung 3.2 wird diese Vorgehensweise schematisch dargestellt.

Wird eine *collapse*-Operation col_v auf eine Kante, die durch die beiden Vertizes v_t und v_u definiert ist, angewendet, verschwindet diese im Vertex v. Die Position von v wird üblicherweise in Abhängigkeit von den angrenzenden Dreiecken f_n ermittelt. Da sich die entsprechenden Berechnungen jedoch anwendungsspezifisch umsetzen lassen, wird hier auf eine detaillierte Erläuterung verzichtet. Nach der Durchführung von col_v können die

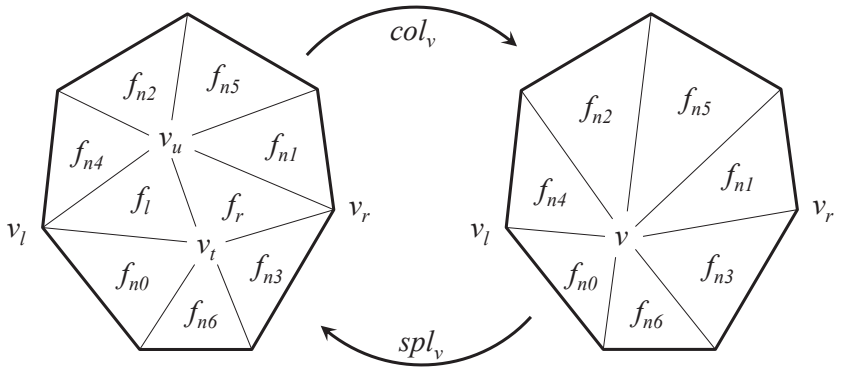


Abbildung 3.2: Zusammenhang zwischen einer *collapse*- und einer *split*-Operation.

degenerierten Dreiecke f_l und f_r, welche direkt an der Kante zwischen v_t und v_u anliegen, entfernt werden. Da die *split*-Operation spl_v der Umkehrung von col_v entspricht, müssen die Dreiecke f_l bzw. f_r während der Aufspaltung von v in v_t und v_u reproduziert werden.

Nach den geometrischen Modifikationen muss die Konnektivität zwischen den anliegenden Dreiecken f_n und den Vertizes v_t, v_u bzw. v aktualisiert werden, um in folgenden Berechnungsphasen Fehler zu vermeiden.

3.2 Verwandte Arbeiten

Im Bereich des Echtzeit-Renderings polygonaler Modelle ist die *Mesh*-Simplifizierung eine der fundamentalen Techniken. Aus diesem Grund existiert eine Vielzahl von Methoden, deren Fokus in erster Linie auf die korrekte Auswertung des Simplifizierungsfehlers ausgerichtet ist. Eine detaillierte Abhandlung über vorhandene Simplifizierungsalgorithmen wurde von Luebke [Lue01] verfasst. Die folgenden Ausführungen beschränken sich auf die Algorithmen zur Echtzeit-Simplifizierung, die denen im Rahmen dieser Arbeit entwickelten Ansätze ähnlich sind.

Das Editieren von *Meshes* stellt ein weiteres wichtiges Forschungsfeld der Grafikprogrammierung dar. Frühere Verfahren waren lediglich in der Lage, Modelle mit einfachen und glatten Strukturen zu verarbeiten, wohingegen die neueren Techniken auch Rücksicht auf die lokalen Eigenschaften eines Polygonnetzes nehmen. Einer der ersten Algorithmen wurde von Welch und Witkin [WW94] entwickelt, welcher die Modellierung sowie die Deformation beliebiger Dreiecksnetze ermöglicht (engl. *free form shape design*). Später wurde dieser Ansatz von Taubin [Tau95] aus Gründen der Effizienz optimiert. Da auf diesem Gebiet weiterhin geforscht wird, sollen im Folgenden die wichtigsten Arbeiten vorgestellt werden.

3.2.1 *Mesh*-Simplifizierung

Rossignac und Borel [RB93] stellten 1993 einen Algorithmus vor, welcher auf der Basis des uniformen *Vertex Clustering* arbeitet. Dabei wird zu einem Modell eine *Bounding Box* erzeugt, welche dieses komplett umschließt und mittels eines regulären Gitters (engl. *regular grid*) sukzessive in mehrere einzelne Zellen unterteilt wird. Anschließend wird zu allen in einer Gitterzelle befindlichen Vertizes der Durchschnitt (engl. *mean*) ermittelt, auf welchen letztlich die komplette Zelle während einer *collapse*-Operation reduziert wird. Durch diese Herangehensweise ist es möglich simplifizierte Variationen eines Modells zu konstruieren. Obwohl mit dem uniformen *Vertex Clustering* relativ schnell Resultate erzielt werden können und die Berechnungen eine präzise obere Schranke für den Simplifizierungsfehler liefern, werden in flachen Modellregionen unnötige Dreiecke platziert, die in einem weiteren Schritt entfernt werden könnten. Low und Tan [LT97] erweiterten diesen Ansatz zu einem gewichteten *Vertex Clustering*, bei welchem Kantenmerkmale, die vom Grid nicht berücksichtigt werden können, erhalten bleiben.

Neben Garland und Heckbert [GH97] führten Popović und Hoppe [PH97] im Jahre 1997 das Prinzip der *Vertex Pair Contraction* ein, bei welchem zwei Vertizes zu einem verschmolzen werden. Auf diesen Ansatz können die meisten aller entwickelten Algorithmen zur *Mesh*-Simplifizierung zurückgeführt werden. In Verbindung mit der metrischen Fehlerquadrik (engl. *quadric error metric*) veröffentlichten Garland und Heckbert eine flexible Möglichkeit, mit welcher sowohl der geometrische Fehler während der Durchführung einer Operation überwacht, als auch die optimale Position des Zielvertex bestimmt werden kann. Später wurde dieses Verfahren hinsichtlich der Verarbeitung einer beliebigen Anzahl von Vertexattributen erweitert [GH98]. Im Vergleich zum *Vertex Clustering* wirkt die generierte Approximation eines Modells bei gleicher Anzahl von Dreiecken qualitativ hochwertiger, wogegen die Performanz signifikant geringer ausfällt. Im Gegensatz dazu können alle benötigten LODs während einer einzigen Simplifizierungssequenz, ausgehend vom Original bis zur gröbsten Stufe, erzeugt werden.

Lindstrom [Lin00] kombinierte mit seinem Verfahren schließlich das *Vertex Clustering* mit den Fehlerquadriken (siehe Abschnitt 3.4.1), um die Positionierung der zu einem Punkt kollabierten Zellen zu verbessern. Nichtsdestotrotz bleibt die hohe Anzahl irrelevanter Dreiecke innerhalb flacher Regionen bestehen. Um dieses Problem zu beseitigen verwendeten Shaffer und Garland [SG01] anstelle des uniformen Gitters einen BSP-Baum, was in einer um den Faktor 3 höheren Laufzeit resultierte. Im Verhältnis zur Simplifizierung mittels *Vertex Pair Contraction* ist dieser Ansatz dennoch schneller, da hier nicht paarweise zwei Vertizes, sondern alle in einer Zelle befindlichen Vertizes kollabiert werden und somit für die entsprechenden Überprüfungen bzw. Kostenberechnungen weniger Zeit in Anspruch genommen wird. Ein adaptives *Vertex Clustering* mithilfe von Octrees wurde später von Schaefer und Warren [SW03a] publiziert. Vergleichsweise kann hier gegenüber der Verwendung von BSP-Bäumen zwar ein Anstieg der Laufzeit verzeichnet werden, allerdings ist die Qualität

der simplifizierten Modelle ähnlich der mittels einer *Vertex Pair Contraction* generierten Approximation. Um die Vorteile des *Vertex Clustering* mit denen des Kollabierens von Kanten verbinden zu können, entwickelten Garland und Shaffer [GS02] einen Algorithmus, dessen Laufzeit unter der eines Verfahrens liegt, welches sich allein auf die Ausführung von *collapse*-Operationen stützt, aber lediglich für die Konstruktion einer einzelnen Detailstufe verwendet werden kann. DeCoro und Tatarchuk [DT07] veröffentlichten eine parallele, auf der GPU lauffähige Variante des *Vertex Clustering*. Diese erweitert die Vorgaben von Schaefer und Warren [SW03a] insoweit als dass diese eine effiziente Datenstruktur bereitstellt, mit welcher der Octree auf einer GPU verarbeitet werden kann. Bei einer sehr hohen Performanz bleiben die typisch qualitativen Probleme aller *Vertex Clustering* Verfahren jedoch bestehen.

3.2.2 Multiskalen-Modellierung

Sollen während einer Modellierungsphase die restlichen geometrischen Details eines *Meshes* beibehalten werden, sind für das Editieren des Modells spezielle Techniken erforderlich. Für derartige Verfahren werden oftmals Multiskalen-Repräsentationen eines Modells verwendet, wobei die Details relativ zu einem *Base Mesh* gespeichert werden und für die einzelnen Qualitätslevel abrufbar sind [FB88, ZSS97, KCVS98, KVS99, GSS99, Gar99]. Wird von einem Anwender ein *Mesh* auf einem niedrigen Qualitätslevel bearbeitet, können im Anschluss die getätigten Modifikationen automatisch zu der nächstfeineren Detailstufe propagiert werden. Durch diese Herangehensweise müssen lediglich wenige Vertizes editiert werden, um größere Veränderungen zu erzielen. Werden die Details einer Modelloberfläche durch *Laplace-* bzw. *Differential-Operatoren* [LSCO*04, SCOL*04] repräsentiert, können die feineren Oberflächenstrukturen mithilfe einer *Laplace-Transformation* bzw. mittels eines linearen *Differentialgleichungssystems* unter Berücksichtigung der modifizierten *Differentialkoeffizienten* rekonstruiert werden.

Zur Manuipulation eines *Meshes* wurde von Sorkine [SCOL*04] ein Editor implementiert, mit welchem zusätzliche Funktionen bereitgestellt werden, wie beispielsweise die interaktive Deformation beliebiger Objektformen (engl. *free form deformation*) innerhalb einer beeinflussbaren Modellregion (engl. *region of influence* - ROI) oder die Möglichkeit, Teile einer Oberfläche in eine andere zu integrieren. Die jeweiligen Transformationen werden auf einer *laplaceschen* Repräsentation des geladenen Modells durchgeführt. Marinov et al. [MBK07] transferierte ein Framework zur Multiskalen-Deformation (engl. *multi resolution deformation* - MRD) auf die GPU, welches keine direkten Transformationen an einem Polygonnetz zulässt. Da der aus einer Operation resultierende Ersatzvektor (engl. *displacement vector*) für jede Komponente unabhängig voneinander kodiert und Details nicht uniform auf der Oberfläche beibehalten werden, können sichtbare Artefakte in hochgradig deformierten Modellregionen ausgemacht werden.

3.3 Motivation und Ziele

Aus den betrachteten Forschungsarbeiten lässt sich zusammenfassend ableiten, dass auf dem Gebiet der *Mesh*-Simplifizierung viele verschiedene Ansätze existieren, welche für die CPU konzipiert und auf vorhandene Begebenheiten hin optimiert wurden. Da derartige Programme lediglich auf der CPU durchgeführt werden können, ist es nahezu unmöglich, Echtzeit-Frameraten (mindestens 25 Frames pro Sekunde) während der Simplifizierung eines Modelles zu erzielen. Bei den Versuchen Simplifizierungsverfahren auf die GPU zu übertragen, wie beispielsweise der von DeCoro und Tatarchuk [DT07] publizierte Algorithmus, handelt es sich um Techniken, die auf dem Prinzip des *Vertex Clustering* beruhen. Mit ihnen können in kürzester Zeit unterschiedliche Detailgrade eines dreidimensionalen Objektes erzeugt werden. Jedoch enthalten die Resultate viele irrelevante Dreiecke, sodass der für die Daten beanspruchte Speicher unnötigerweise erhöht wird. Werden entsprechende Dreiecke aus dem *Mesh* entfernt, hätte dies keinen Detailverlust zur Folge. Das Aufkommen belangloser Dreiecke hängt zum Teil damit zusammen, dass nach und nach immer nur eine einzelne Zelle kollabiert und die Möglichkeit des Kontrahierens von in zwei benachbarten Zellen befindlichen Teilmengen außer Acht gelassen wird.

Das Ziel der eigenen Forschungsarbeit ist es, ein Simplifizierungsverfahren für die GPU zu entwickeln, mit welchem es möglich wird mehrere Detailstufen eines dreidimensionalen Modells in Echtzeit zu konstruieren. Hierfür muss eine effiziente Datenstruktur konstruiert werden, mit welcher die eingelesen Objektdaten parallel auf der Grafikkarte verarbeitet und anschließend für weitere Anwendungsszenarien aufbereitet werden können. Zudem soll die erste GPU-Implementierung im Bereich der *Vertex Pair Contraction* realisiert werden, welche aller Voraussicht nach langsamer als eine GPU-Umsetzung zum *Vertex Clustering* sein wird, jedoch für die Generierung der einzelnen Detailstufen, bei einer vergleichbaren Qualität, weniger Dreiecke benötigt und somit den Datenoverhead reduziert. Die Minimierung der Dreiecke wird dadurch ermöglicht, da vor dem Kollabieren einer Kante zunächst sämtliche Möglichkeiten analysiert werden, über welche ein Vertexpaar zu einem Punkt kontrahiert werden kann. Diese Überprüfung sorgt gegenüber dem *Vertex Clustering*, bei dem alle Vertizes einer Zelle einfach miteinander verschmolzen werden, ohne dabei den Bezug zu einer benachbarten Zelle zu untersuchen, einerseits für eine geringere Performanz, auf der anderen Seite allerdings für bessere Ergebnisse.

Für die Modellierung dreidimensionaler Objektstrukturen existiert derzeit noch kein nennenswertes Verfahren, mit welchem die Modelle interaktiv (10 bis 15 Frames pro Sekunde) deformiert und entsprechende Berechnungen parallel auf der GPU vollzogen werden. Marinov et al. [MBK07] stellen zwar ein adäquates Framework zur Verfügung, die vom Anwender vorgenommenen Modifikation werden jedoch nicht direkt auf den geladenen Modellen ausgeführt, was zu wahrnehmbaren Fehlern führt. Des Weiteren ist es oftmals nicht möglich ein Modell auf mehreren Auflösungsstufen zu manipulieren und die jeweiligen Veränderungen zwischen den verschiedenen Leveln zu propagieren. Die

Fähigkeit zwei Modelle oder lediglich einige ausgewählte Teile zu kombinieren, wie es mithilfe des von Sorkine [SCOL*04] vorgestellten Editors möglich ist, kann in Hinblick auf die Simplifizierung und dem Umschalten zwischen verschiedenen Detailgraden sowie bei der Verwaltung der Modellveränderungen nur schwerlich verwirklicht werden. Hierfür sind Anpassungen an der Topologie eines Modelles vonnöten, über welche entweder Daten zur Geometrie in eine bestehende Objektstruktur integriert oder eben aus dieser entfernt werden müssen. Bei einer derartigen Aktualisierungsphase sollten die Modelldaten immer auf dem höchsten verfügbaren Detailgrad miteinander verknüpft, als eigenständiges Modell angesehen und enstprechend neu simplifiziert werden.

Somit stellt sich die Aufgabe, ein auf der GPU lauffähiges Programm zu erstellen, mit welchem ein Modell geladen, simplifiziert und modifiziert werden kann. Um dies zu ermöglichen muss eine angemessene Datenstruktur erstellt werden, mit welcher sowohl die zu dem Modell gehörenden Detailstufen parallel erzeugt und die vom Anwender vollführten Deformationen in einer korrekten Art und Weise umgesetzt bzw. über die verschiedenen Detailgrade hinweg konsistent weitergeleitet und parallel verarbeitet werden können.
Zusammenfassend können folgende Ziele festgehalten werden:

- der Entwurf eines hochqualitativen, parallelen Simplifizierungsalgorithmus, basierend auf dem Prinzip der *Vertex Pair Contraction*
- die effiziente Berechnung der optimalen Vertexposition, in welche eine durch ein Vertexpaar definierte Kante kollabiert
- die Verwendung eines adäquaten Fehlermaßes zur Minimierung des Simplifizierungsfehlers
- die Simplifizierung von hochgradig detaillierten Modellen in weniger als einer Sekunde
- das interaktive Editieren von großen Datenmodellen (10 bis 15 Frames pro Sekunde)
- die in Echtzeit vollzogene Propagation der Modifikationen über alle Detailstufen hinweg (mindestens 25 Frames pro Sekunde)
- die Generierung einer hinreichenden, für die Modellierung notwendigen Datenstruktur während der parallelen *Mesh*-Simplifizierung
- die Entwicklung einer Datenstruktur, mit welcher Modelle auf verschiedenen Detailstufen bearbeitet und zur Erstellung von Animationen verwendet werden können

Im Folgenden wird ein selbst entwickeltes Verfahren zur Simplifizierung von 3D-Modellen vorgestellt, welches entsprechende Berechungen parallel auf dem Grafikprozessor durchführt. Anschließend wird eine GPU-Implementierung zur Modellierung dreidimensionaler Objekte erläutert. Die spezifischen algorithmischen Abläufe werden dabei im Zusammenhang mit den neu entworfenen, auf den jeweiligen Anwendungsfall ausgerichteten Datenstrukturen

betrachtet und abschließend einigen Tests unterzogen, um Aussagen darüber treffen zu können, ob die oben genannten Ziele erreicht wurden.

3.4 Simplifizierung von 3D-Modellen

In diesem Kapitel soll eine Möglichkeit für das parallele Simplifizieren von komplexen 3D-Modellen diskutiert werden. Hochgradig detaillierte Modelle werden vor allem im Bereich der Computerspiele oder anderen interaktiven Rendering-Applikationen eingesetzt. In diesem Kontext greifen die Entwickler regelmäßig auf statische Level-of-Detail Techniken zurück, um Echtzeit-Frameraten gewährleisten zu können. Dies ist zwar eine einfache Strategie zur Erhöhung der Rendering-Performanz, jedoch müssen dabei die zusätzlichen Detailstufen bzw. die für die nächstfeineren Level fehlenden Geometriedaten gespeichert und bei Bedarf auf die Grafikkarte übertragen werden. Problematisch ist dies vor allem bei Online-Anwendungen, bei welchen die Daten über eine möglicherweise schwache Netzwerkverbindung zum Client hin transferiert werden müssen.

Die Entwicklung von effizienten Simplifizierungsverfahren ist eines der wichtigsten Forschungsgebiete in der Computergrafik. Mithilfe einer Simplifizierungsmethode können automatisch die verschiedensten Detailgrade zu einem 3D-Objekt generiert werden, sodass lediglich die höchste Modellstufe bis ins Detail von einem Grafikdesigner erstellt werden muss. Die Objektoberfläche inklusive der feinen Strukturen wird überlicherweise durch ein Dreiecksnetz (*Mesh*) repräsentiert, welches zu Darstellungszwecken gerendert werden muss. Da derartige *Meshes* aufgrund ihrer Komplexität enorme Datenmengen produzieren, besteht die Hauptaufgabe eines Simplifizierungsverfahrens darin, die Anzahl der zu rendernden Dreiecke zu reduzieren ohne visuell erkennbare Spuren zu hinterlassen. Um die Differenzen zwischen einer erzeugten Detailstufe und dem Original analysieren zu können, muss ein Fehlermaß für den entsprechenden Simplifizierungsfehler ε definiert werden. Wird dieser minimiert, können die visuellen Unterschiede eingegrenzt werden. Für die Parallelisierung der einzelnen Berechnungsschritte muss eine intelligente Datenstruktur konzipiert werden, in welcher sich alle für die Simplifizierung relevanten Daten befinden.

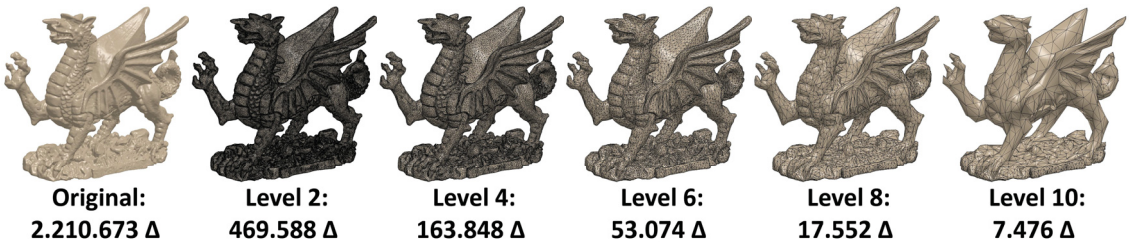


Abbildung 3.3: Das Originalmodell des *Welsh Dragon* sowie einige der mithilfe des parallelen Simplifizierungsverfahrens generierten Detailstufen. Insgesamt wurden 10 Level innerhalb von 0.73 Sekunden erzeugt.

Heutzutage verfügen viele Rechner über eine Grafikkarte, mit deren Hilfe parallele Berechnungen vollzogen werden können. Aufgrund dieser Tatsache wurde ein hochqualitativer, paralleler, auf den metrischen Fehlerquadriken basierender Simplifizierungsalgorithmus entwickelt [GDG11], mit welchem mehrere Detailstufen eines 3D-Modells in weniger als einer Sekunde erzeugt werden können (siehe Abbildung 3.3). Demnach kann die Simplifizierungsgeschwindigkeit durchaus mit der Ladezeit von zusätzlichen, auf der lokalen Festplatte befindlichen Modelldaten konkurrieren. Die Grundidee des Verfahrens beruht dabei auf dem Kollabieren von Kanten. Da während der Simplifizierung keine besondere Sortierung der Kanten berücksichtigt werden muss, können die entsprechenden Operationen parallel ausgeführt werden. Die Fehleranalyse sowie die Lokalisierung des Zielvertizes wird über die Auswertung von Vertexquadriken realisiert, welche innerhalb einer effizienten Datenstruktur gehalten werden.

Der im Folgenden beschriebene Algorithmus wird im Kapitel 3.5 um die Möglichkeit zur Multiskalen-Modellierung (engl. *multi resolution modeling* - MRM) erweitert. Für die eindeutige Differenzierung wird die hier diskutierte Vorgehensweise im späteren Verlauf deshalb als *Basis-Simplifizierer* bezeichnet.

3.4.1 Metrische Fehlerquadriken (*Quadric Error Metric*)

Wird bei einer *collapse*-Operation col_v eine Kante e, welche durch die beiden Vertizes v_t und v_u definiert ist, zu einem Punkt v kollabiert, muss die Position von v in Abhängigkeit von den Kosten für col_v ermittelt werden. Die Kosten wiederum müssen minimiert werden, wenn der Fehler bzw. die Diskrepanz zwischen Orignalmodell und simplifiziertem *Mesh* möglichst gering ausfallen soll. Um eine akkurate Kontrolle über den geometrischen Fehler, der durch die Simplifizierung eines Modells ausgelöst wird, gewährleisten und zudem die Kosten von col_v entsprechend evaluieren zu können, muss für das Simplifizierungsverfahren ein geeignetes Fehlermaß gewählt werden. Für die Algorithmen zur Simplifizierung von Polygonnetzen bzw. *Meshes*, welche auf das Kollabieren von Kanten zurückgeführt werden können, publizierten Garland und Heckbert [GH97] die metrischen Fehlerquadriken (engl. *quadric error metric*), durch welche der Ähnlichkeitsgrad zwischen einem simplifizierten Modell und seinem Original abgeschätzt werden kann. Die Approximation basiert dabei auf der Berechnung der Differenz zwischen einem simplifizierten Vertex zu den Ebenen, welche sich aus den angrenzenden Dreiecken des Originals ergeben. Sei $P(v)$ eine Menge von Ebenen, die an einem Vertex v anliegen, so kann der maximale Fehler mithilfe der Summe der quadratischen Differenzen (engl. *sum of squared distances* - SSD) ermittelt werden:

$$\Delta(\mathbf{v}) = \Delta[v_x \; v_y \; v_z \; 1]^T = \sum_{\mathbf{p} \in P(v)} (\mathbf{p}^T \mathbf{v})^2, \tag{3.1}$$

wobei durch $\mathbf{p} = [a \; b \; c \; d]^T$ die implizite Ebenengleichung $ax + by + cz + d = 0$ in normalisierter Form repräsentiert wird. Die Werte von a, b und c entsprechen dabei der

Normalen von $\mathbf{p}$ und d stellt die vorzeichenbehaftete Distanz zum Koordinatenursprung dar. Die Formel 3.1 kann in die folgende quadratische Form überführt werden:

$$\begin{aligned} \Delta(\mathbf{v}) &= \sum_{\mathbf{p}\in P(v)} (\mathbf{v}^T\ \mathbf{p})(\mathbf{p}^T\ \mathbf{v}) && (3.2)\\ &= \sum_{\mathbf{p}\in P(v)} \mathbf{v}^T\ (\mathbf{p}^T)\ \mathbf{v} && (3.3)\\ &= \mathbf{v}^T \left(\sum_{\mathbf{p}\in P(v)} \mathbf{Q}_p \right) \mathbf{v}, && (3.4) \end{aligned}$$

wobei $\mathbf{Q}_p$ mit der Kovarianzmatrix einer Ebene $\mathbf{p}$ aus $P(v)$ gleichzusetzen ist.

Um einer frühzeitigen Degeneration der Meshgrenzen vorzubeugen, kann pro Randkante neben der Vertexquadrik eine Randquadrik betrachtet werden, bei der eine zur Dreiecksebene $\mathbf{p}$ orthogonal verlaufende virtuelle Ebene berücksichtigt wird. Seien v_1 und v_2 die Vertizes einer Randkante und v_3 der dritte Vertex des einzigen, an der Randkante anliegenden Dreiecks, so ist die Normalengleichung der virtuellen Ebene wie folgt definiert:

$$\begin{aligned} t_x x + t_y y + t_z z - n \cdot v_1 &= 0 \qquad mit && (3.5)\\ t &= \frac{e_2 - (e_1 \cdot e_2)e_1}{\|e_2 - (e_1 \cdot e_2)e_1\|}, && (3.6)\\ e_1 &= \frac{v_2 - v_1}{\|v_2 - v_1\|}, && (3.7)\\ e_2 &= \frac{v_3 - v_1}{\|v_3 - v_1\|}. && (3.8) \end{aligned}$$

In diesem Zusammenhang lässt sich die metrische Fehlerquadrik für beliebige Dimensionen laut [GH98] verallgemeinern. Die allgemeine Quadrik $\mathbf{Q}_p$ lautet demnach:

$$\mathbf{Q}_p = \begin{pmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & d \end{pmatrix}, \tag{3.9}$$

wobei

$$\begin{aligned} \mathbf{A} &= \mathbf{Id} - e_1 e_1^T - t t^T && (3.10)\\ \mathbf{b} &= (v_1 \cdot e_1)e_1 + (v_1 \cdot t)t - v_1 && (3.11)\\ d &= v_1 \cdot v_1 - v_1 \cdot e_1 - v_1 \cdot t && (3.12) \end{aligned}$$

Im Bezug auf die Kostenberechnung einer *collapse*-Operation, bei welcher das Vertexpaar v_t und v_u zu v zusammengefasst wird, kann der zugehörige Fehler $\overline{\mathbf{Q}}$ aus den entsprechenden Vertexquadriken $\mathbf{Q}_{v_t}$ und $\mathbf{Q}_{v_u}$ abgeleitet werden. Die Summe der quadratischen Differenzen (engl. *sum of squared differences* - SSD) lässt sich mithilfe von $\overline{\mathbf{Q}} = \mathbf{Q}_{v_t} + \mathbf{Q}_{v_u}$ beschreiben und kann zusätzlich zur Bestimmung der optimalen Position von v ausgenutzt werden. Für

ein derartiges Optimum muss die SSD jedoch minimiert werden, wofür letztlich folgendes lineare Gleichungssystem gelöst werden muss:

$$\nabla\overline{\mathbf{Q}}\ (\overline{v}) = \begin{pmatrix} \mathbf{A} & \mathbf{b} \\ 0\cdots0 & 1 \end{pmatrix} \begin{pmatrix} \overline{v} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}. \tag{3.13}$$

3.4.2 Generierung statischer LODs

Sollen zu einem gegebenen dreidimensionalen Modell statische LODs generiert werden, müssen für die einzelnen Detailstufen sukzessive Modifikationen an der Geometrie des Modells vorgenommen werden. In der Regel werden dabei Kanten, die durch zwei Vertizes v_t und v_u definiert sind, zu einem Punkt kollabiert, wodurch die an den Kanten anliegenden Dreiecke (auch als *faces* bezeichnet) verschwinden. Werden derartige *collapse*-Operationen ausgeführt, muss dabei beachtet werden, dass die grundlegende Form des Modells bzw. die dort enthaltenen Strukturen bis zu einem gewissen Grad erhalten bleiben. Es müssen also zu jeder Operation die Kosten und der durch sie erzeugte Fehler bestimmt und minimiert werden. Wird nun eine Kante zu einem Punkt kollabiert, ist dessen Position entscheidend für die Qualität der LODs. Aus diesem Grund sollte die Berechnung der optimalen Vertexposition bei der Kostenminimierung berücksichtigt werden. Zur Vereinfachung wird jedoch oft auf den Mittelpunkt der durch v_t und v_u gegebenen Strecke zurückgegriffen oder einer der beiden Vertizes v_t bzw. v_u als Ziel des Kollabierungsvorgangs angenommen. Dies kann zu Simplifizierungsfehlern führen, welche sich über die einzelnen Detailstufen hinweg fortpflanzen können. Für den hier vorgestellten Algorithmus [GDG11], welcher zu verhältnismäßig äquivalenten Anteilen von Dipl. Inf. Evgenij Derzapf mitentwickelt wurde, werden die von Garland und Heckbert eingeführten *Quadric Error Metrics* [GH98] zur Kostenberechnung verwendet. Diese führen zwar zu einer Überbewertung des Fehlers, liefern jedoch die optimale Position des Vertex, zu dem eine Kante kontrahiert wird.

Da die Simplifizierung auf einer parallelen Ausführung von *collapse*-Operationen beruht, deren Kosten über die *Quadric Error Metrics* abgeschätzt werden, wird eine adäquate Datenstruktur benötigt, welche zum einen alle für die Parallelisierung relevanten Informationen beinhaltet und zum anderen den Speicherverbrauch auf ein Minimum reduziert. In diesem Sinne kann der Algorithmus in zwei separate Phasen untergliedert werden:

- Aufbau der Datenstruktur
- Parallele Simplifizierung

Die algorithmischen Abläufe der einzelnen Phasen werden in Abbildung 3.4 schematisch veranschaulicht und in den folgenden Abschnitten genauer diskutiert.

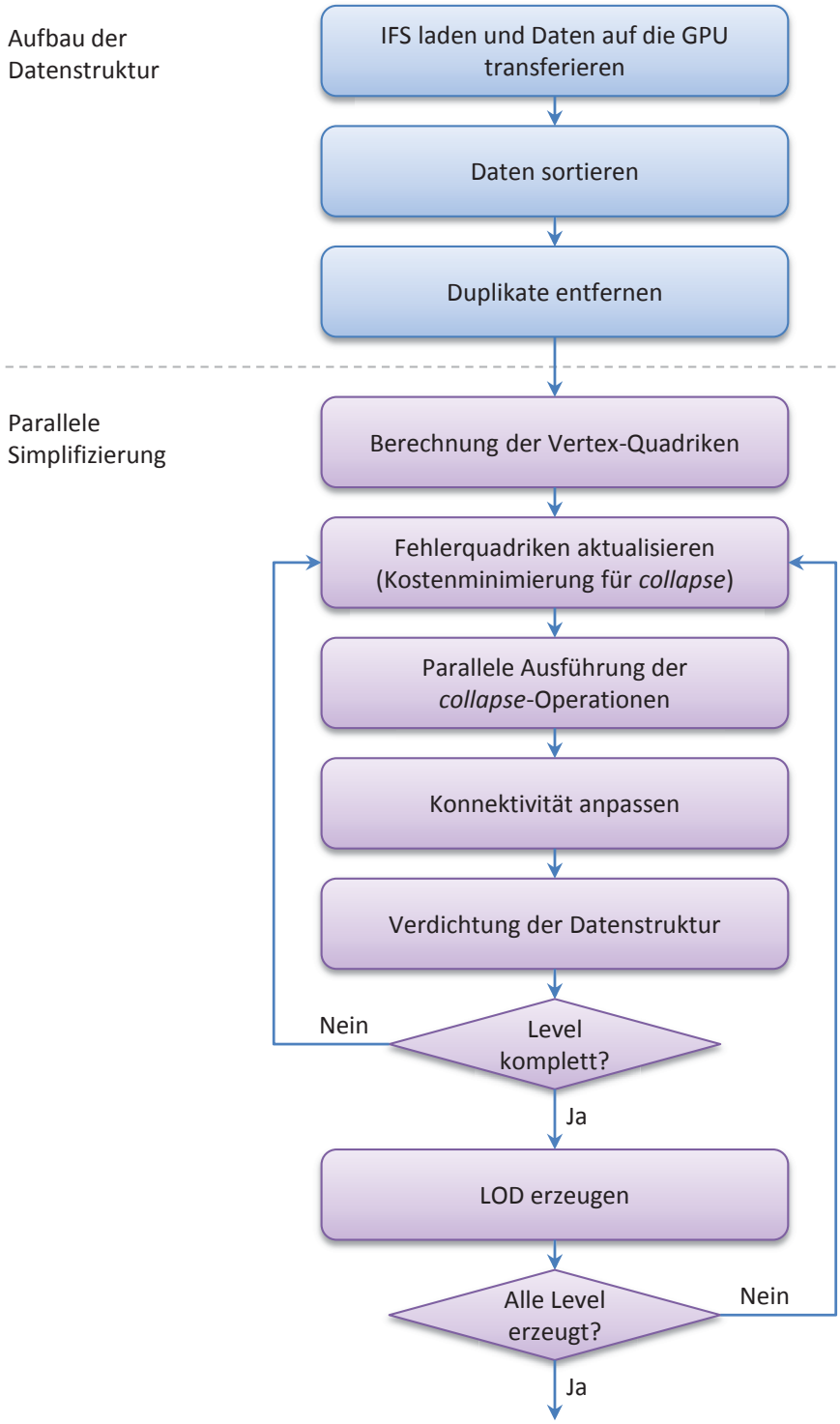


Abbildung 3.4: Verfahrensablauf zur Simplifizierung bzw. zur Generierung von statischen LODs eines dreidimensionalen Polygonnetzes. Der Algorithmus lässt sich in zwei Hauptphasen untergliedern: Der Aufbau der Datenstruktur und deren Verwendung für die parallele Simplifizierung eines 3D-Modells.

01. Strukturaufbau **IFS laden**

3.4.2.1 Aufbau der Datenstruktur

Als Initialschritt wird ein *Indexed-Face-Set* (IFS) geladen und im Anschluss auf die GPU übertragen. Durch das IFS werden die Attribute der Vertizes (Position, Normale usw.) und die zu einem Dreieck gehörenden Vertexindizes i_1, i_2 und i_3 geliefert. Letztere können für die Generierung der benötigten Kanten-Datenstruktur verwendet werden, indem für jedes Dreieck drei Kanten erzeugt werden. Jede Kante (engl. *edge*) ist dabei durch zwei der drei Vertexindizes definiert, wobei die Ordnung $i_1 < i_2 < i_3$ berücksichtigt wird. Um im späteren Verlauf des Verfahrens die Fehlerquadriken der Randkanten anpassen zu können, wird zusätzlich zu jeder Kante der Index des ihr im Dreieck gegenüberliegenden Vertex gespeichert. Ohne eine gesonderte Betrachtung der Randkanten bzw. einer Gewichtung ihrer Quadriken, würden diese bei der Durchführung der *collapse*-Operationen zu früh kollabieren, was zu sichtbaren Simplifizierungsfehlern führen kann.

02. Strukturaufbau **Daten sortieren**

Während dieses Vorgangs werden der Datenstruktur mehrere Kantenduplikate hinzugefügt, die zur Minimierung des Speicherbedarfs bzw. zur Steigerung der Performanz eliminiert werden müssen. Um dies zu bewerkstelligen werden die erzeugten Kanten hinsichtlich ihrer Vertexindizes i_{min} und i_{max} mithilfe von *Radixsort* bzw. *Distributionsort* sortiert.

03. Strukturaufbau **Duplikate entfernen**

Liegen die Kanten in lexikographischer Ordnung vor, können diese miteinander verglichen werden. Entsprechen die zu einer Kante e gehörenden Vertexindizes denen der Vorgängerkante, so handelt es sich dabei um ein Replikat. Im entwickelten Verfahren werden Duplikate mit 0 und alle anderen Kanten mit 1 markiert. Nach dem Markierungsprozess kann mithilfe der Berechnung der Präfix-Summe die Anzahl der voneinander unterscheidbaren Kanten ermittelt werden, sodass ein neuer *Buffer* von entsprechender Größe angelegt und alle Kantenduplikate aus der Datenstruktur entfernt werden können. Dieses Prinzip wird durch Abbildung 3.5 veranschaulicht. Um im Nachhinein Randkanten als solche identifizieren zu können, wird jeder Kante ein zusätzliches Flag zugeordnet, welches während der Beseitigung der Replikate initialisiert wird. Wurde im vorherigen Schritt kein Duplikat zu einer Kante e gefunden, wird das Randkantenflag von e auf 1,

Kanten	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
Duplikat?	N	J	N	N	J	N	J	N	J	N	N	N	J	N	
Flag	1	0	1	1	0	1	0	1	0	1	1	1	0	1	
Präfix-Summe	0	1	1	2	3	3	4	4	5	5	6	7	8	8	9
Randkante?	N	N	J	N	N	N	N	N	N	J	J	N	N	J	
Flag	0	0	1	0	0	0	0	0	0	1	1	0	0	1	

Abbildung 3.5: Basisprinzip zur Markierung von Duplikaten und Randkanten. Mithilfe des Duplikat-Flags ergibt sich die korrekte Kantenanzahl aus der Präfix-Summe.

andernfalls auf den Wert 0 gesetzt.

Die während der Generierung der Datenstruktur vollzogenen Schritte werden in Algorithmus 5 zusammengefasst. Die Datenfelder, welche für den Aufbau der Kantenstruktur

```
foreach face f in parallel do
    i1,i2,i3 = get_face_indices(f)
    edge1 = create_edge(min(i1,i2),max(i1,i2),i3)
    edge2 = create_edge(min(i2,i3),max(i2,i3),i1)
    edge3 = create_edge(min(i3,i1),max(i3,i1),i2)
RADIX SORT edges in parallel by imax
RADIX SORT edges in parallel by imin
foreach edge e in parallel do
    if (get_previous_edge(e) != e)
        set_edge_flag(e, 1)
        if (get_next_edge(e) != e) set_single_flag(e, 1)
        else set_single_flag(e, 0)
    else set_edge_flag(e, 0)
COMPACT edges in parallel
```

Algorithmus 5: Parallele Generation der Kanten-Datenstruktur.

benötigt werden, sowie der entsprechende Speicherverbrauch, wurden in der Tabelle 3.1 vermerkt. Die wichtigsten Komponenten sind der *Vertex* und *Index Buffer*, welche die Attribute sämtlicher Vertizes und die Daten zur Konnektivität enthalten. Zu jeder Kante wird neben den jeweiligen Indizes auch Speicher für den gegenüberliegenden Vertex und für das Flag zur Randkantenbetrachtung alloziert. Dies führt zu einem maximalen Speicherbedarf von $246 + 4k$ Bytes pro Vertex[1], wobei k der Anzahl der Vertexattribute entspricht. Nachdem alle Kantenduplikate entfernt und der temporäre Speicher freigegeben wurde, kann dieser auf $63 + 4k$ Bytes pro Vertex reduziert werden.

buffers	**elements**	**bytes per entry**
faces	index VBO	12
edges	vertex index (×2)	8
	vertex index (opposite)	4
	single edge flag	1
vertices	vertex VBO	$4k$
temporary per edge	sorting edge (×3)	12
	sort order	4
	sort key	4
	sort prefix sum (scan)	4

Tabelle 3.1: Datenstruktur zur Generierung der Kanteninformationen, wobei k der Anzahl der berücksichtigten Vertexattribute entspricht.

1 Da pro Dreieck drei Kanten angelegt werden, kann jede Kante maximal zwei Mal in der Datenstruktur auftauchen. Aus diesem Grund muss der in Tabelle 3.1 vermerkte Speicherbedarf für die Kanten (inkl. temporärem Speicher von 111 Bytes pro Vertex) zunächst doppelt gezählt werden, um den Maximalbedarf zu erhalten. Nach der Entfernung der Duplikate werden nur noch 39 Bytes pro Vertex für die Kanteninformationen benötigt.

3.4.2.2 Parallele Simplifizierung

Für die Parallelisierung des Verfahrens wurde der Algorithmus in mehrere unabhängige Einzelschritte unterteilt. Da für jeden Berechnungsschritt entsprechende Daten vorliegen müssen, ist die Unterteilung zudem für die Datensynchronisation entscheidend. Jeder Schritt stellt somit eine in sich abgeschlossene Einheit dar, welche parallel auf dem Grafikprozessor (GPU) ausgeführt werden kann.

Um die relevanten Zusatzinformation in der Datenstruktur hinterlegen zu können, muss diese um entsprechende Felder erweitert werden. Zur Erzeugung der LODs bleiben der *Vertex* und *Index Buffer* bestehen. Da nach der Ausführung einer *collapse*-Operation die Gültigkeit der involvierten Dreiecke überprüft werden muss, wird ein entsprechendes Flag angelegt. Für jede einzelne Kante werden beide Vertexindizes, ihre Kosten für einen *collapse*, die zugehörige Zielposition (inklusive der berücksichtigten Vertexattribute) und zwei Flags in die Datenstruktur aufgenommen, um entscheiden zu können, ob die *collapse*-Operation durchgeführt werden darf und ob die Kante im Zuge der Operation ungültig wurde. Da die Kosten einer *collapse*-Operation auf die Fehlerquadriken der zu einer Kante gehörenden Vertizes zurückgeführt werden können, müssen pro Vertex Felder zum Speichern der jeweiligen Quadrik, für den Verweis auf die anliegende Kante mit den geringsten Kosten, sowie ein Flag zur Gültigkeitsbestimmung definiert werden. Um nachvollziehen zu können in welchem Punkt ein Vertex während eines Kollabierungsvorgangs verschwunden ist, wird zusätzlich die ID des Zielvertex (engl. *collapse target*) vermerkt.

buffers	**elements**	**bytes per entry**
faces	index VBO	12
	active flag	4
edges	vertex index (×2)	8
	edge cost	4
	collapse state	4
	optimal placement	$4k$
	active flag	4
vertices	vertex VBO	$4k$
	vertex quadric	$2k^2 + 6k + 4$
	min edge ID	4
	min edge cost / collapse target	4
	active flag	4
temporary	sort prefix sum (scan)	4

Tabelle 3.2: Elemente der für die Simplifizierung genutzten Datenstruktur, wobei k der Anzahl der berücksichtigten Vertexattribute entspricht.

Die Elemente der verwendeten Datenstruktur wurden in Tabelle 3.2 zusammengefasst. Das temporäre Feld ist für die Verdichtung der Datenstruktur notwendig, wenn Kanten ungültig und aufgrund dessen aus der Datenstruktur entfernt werden können. Da die

verwendeten Quadriken Symmetrieeigenschaften aufweisen, kann durch das Speichern der oberen Dreiecksmatrix der Speicherbedarf für die Vertexquadriken von $4k^2 + 8k + 4$ auf $2k^2 + 6k + 4$ reduziert werden, wobei k der Anzahl an Vertexattributen entspricht.

01. Para. Simplifizieren **Vertexquadriken berechnen**

Nachdem in der ersten Phase des Algorithmus die grundlegende Datenstruktur aufgebaut wurde, müssen für die Kostenabschätzung einer *collapse*-Operation die zugehörigen Vertexquadriken ermittelt werden. Pro Vertex entspricht dies der Summe aller angrenzenden Dreiecksquadriken (siehe Abschnitt 3.4.1). Im Bezug auf die Parallelisierung des Verfahrens kann der Algorithmus an dieser Stelle parallel über alle Dreiecke laufen und die Quadrik der einzelnen Vertizes bestimmen und aufsummieren. Da sich letztere im weiteren Verlauf der parallelen Simplifizierung nicht mehr ändern, können sie vorab ermittelt und in der Datenstruktur hinterlegt werden.

Um das frühzeitige Kollabieren von Randkanten zu vermeiden, werden die korrespondierenden Quadriken zusätzlich gewichtet. Diese Anpassung wird unter Berücksichtigung des Vertex vollzogen, welcher sich gegenüber der Randkante befindet. Zur Vereinfachung wurde der entsprechende Index bereits beim Anlegen der Datenstruktur gesichert. Genauere Details zur Aktualisierung bzw. Berechnung der Vertexquadriken wurden im Abschnitt 3.4.1 besprochen. Der Algorithmus 6 soll diese Vorgehensweise nochmals verdeutlichen.

```
foreach face f in parallel do
    i1,i2,i3 = get_face_indices(f)
    quad = compute_face_quadric(f)
    add_quadric(i1, quad)
    add_quadric(i2, quad)
    add_quadric(i3, quad)
foreach single_edge e in parallel do
    i3 = get_opposite_vertex(e)
    quad = compute_boundary_quadric(imin[e], imax[e], i3)
    add_quadric(imin[e], quad)
    add_quadric(imax[e], quad)
```

Algorithmus 6: Parallele Berechnung der Vertexquadriken.

02. Para. Simplifizieren **Fehlerquadriken aktualisieren**

Im ersten Schritt der Simplifizierungsschleife werden die Kosten für das Kollabieren einer Kante e und anschließend die innerhalb einer Iteration durchführbaren *collapse*-Operationen ermittelt. Eine Kante e darf kollabieren, wenn die zugehörigen Kosten maximal ε^2 betragen, wobei ε den Simplifizierungsfehler repräsentiert. Befinden sich die Kosten unterhalb der angegebenen Schranke, wird die Kante als möglicher *collapse*-Kandidat markiert.

Für die Berechnung der Kosten, werden die Quadriken $Q_{v_{min}}$ und $Q_{v_{max}}$ der beiden Kantenvertizes v_{min} bzw. v_{max} zur Kantenquadrik $\overline{Q}$ addiert. Mithilfe von $\overline{Q}$ kann ein lineares Gleichungssystem aufgestellt werden, dessen Lösung der optimalen Position des *collapse*-Vertex $\overline{v}$ entspricht. Da es sich bei der in $\overline{Q}$ enthaltenen Teilmatrix A um ei-

ne symmetrische, positiv semi-definite Matrix handelt, kann die LDL-Dekomposition (Cholesky-Dekomposition ohne Quadratwurzeln) verwendet werden, um das entsprechende Gleichungssystem (siehe Formel 3.13) aufzulösen. Wurde die optimale Position von $\overline{v}$ ermittelt, können in Verbindung mit $\overline{Q}$ die Kosten für den *collapse* evaluiert werden. Während der Evaluation wird für die beiden Vertizes v_{min} und v_{max} die Kante mit den minimalen Kosten bestimmt und gespeichert. Durch Algorithmus 7 wird die parallele Minimierung der Fehlerquadriken veranschaulicht.

```
foreach edge e in parallel do
    quad = calc_edge_quadric(vertex_quadric[v_min],vertex_quadric[v_max])
    collapse_pos[e] = optimize_pos(quad)
    edge_cost[e] = calc_cost(quad, collapse_pos[e])
    if edge_cost[e] ≤ ε²
        collapse_state[e] = collapse
        atomic_min(min_edge_cost[v_min], edge_cost[e])
        atomic_min(min_edge_cost[v_max], edge_cost[e])
    else collapse_state[e] = no_operation
```

Algorithmus 7: Parallele Berechnung bzw. Minimierung der Fehlerquadriken.

03. Para. Simplifizieren ***Collapse*-Operationen ausführen**

Wurde für jeden Vertex die angrenzende Kante mit den niedrigsten *collapse*-Kosten gefunden, kann mit der Ausführung der *collapse*-Operationen begonnen werden. Das Kollabieren einer Kante e ist nur dann möglich, wenn deren Kosten dem lokalen Minimum entsprechen. Um dies sicherzustellen werden die für die der Kante e zugehörigen Vertizes v_{min} und v_{max} hinterlegten Referenzen verglichen. Sind die Referenzen, welche jeweils auf die anliegende Kante mit den geringsten Kosten verweisen, identisch bzw. zeigen sie beide auf die aktuell betrachtete Kante e, kann davon ausgegangen werden, dass in der direkten Nachbarschaft von e keine weitere Kante existiert, deren Kosten niedriger sind. Das lokale Minimum wurde demnach gefunden und die entsprechende *collapse*-Operation darf vollzogen werden. Wird hingegen keine Übereinstimmung gefunden, muss der Status von e als *collapse*-Kandidat zurückgesetzt werden.

```
foreach edge e in parallel do
    if collapse_state[e] == collapse
        cost = get_edge_cost(e)
        if min_edge_cost[v_min] == cost: edge_ID[v_min] = e
        if min_edge_cost[v_max] == cost: edge_ID[v_max] = e
foreach edge e in parallel do
    if collapse_state[e] == collapse
        if edge_ID[v_min] == e and edge_ID[v_max] == e
            v̄ = collapse_pos[e]
            vertex_quadric[v_min] += vertex_quadric[v_max]
            collapse_target[v_max]=v_min
            vertex_active[v_max]=false
```

Algorithmus 8: Parallele Ausführung der *collapse*-Operationen.

Nachdem alle gültigen *collapse*-Operation herausgefiltert wurden, können diese parallel ausgeführt werden. Dabei wird der Vertex v_{min} auf die zuvor berechnete Position von $\overline{v}$ verschoben und $\overline{Q}$ als neue Vertexquadrik angenommen. Die Vertexattribute von $\overline{v}$ sind zusammen mit der korrespondierenden Fehlerquadrik $\overline{Q}$ für jeden *collapse*-Kandiaten in der Kantendatenstruktur gesichert worden. Der Kantenvertex v_{max} wird durch diese Herangehensweise ungültig, entsprechend markiert und v_{min} als Zielvertex vermerkt. Letzteres ist für eine effiziente Detektion von degenerierten Kanten und Dreiecken vonnöten, wenn diese in einem späteren Schritt aus der Datenstruktur entfernt werden. Die parallele Ausführung der *collapse*-Operationen wird in Algorithmus 8 dargestellt.

04. Para. Simplifizieren **Konnektivität anpassen**

Wurden alle innerhalb einer Iteration möglichen *collapse*-Operationen vollzogen, muss die Konnektivität zwischen den einzelnen Komponenten der Datenstruktur aktualisiert werden. Dreiecke sowie Kanten, in deren direktem Umfeld eine Kante e kollabiert wurde, referenzieren noch immer die ursprünglichen Kantenvertizes v_{min} und v_{max}. Diese Referenzen müssen durch einen Verweis auf den *collapse*-Vertex $\overline{v}$ ersetzt werden (siehe Abbildung 3.2). Da die Daten von $\overline{v}$ (Attribute und Vertexquadrik) während der *collapse*-Operation auf v_{min} übertragen werden, müssen lediglich die Verweise auf v_{max} erneuert werden. Hierbei ist der zuvor eingetragene Vermerk auf den Zielvertex von v_{max} (das *collapse target*) hilfreich. Ist ein Dreieck oder eine Kante während der Durchführung einer Operation verschwunden, können diese als ungültig deklariert und in einem weiteren Schritt entfernt werden. Dieser Sachverhalt wird durch Algorithmus 9 zusammengefasst.

```
foreach face f  in parallel do
    update_indices(f,collapse_target)
    if degenerate(f): face_valid[f]=false
foreach edge e  in parallel do
    update_indices(e,collapse_target)
    if degenerate(e): edge_valid[e]=false
```

Algorithmus 9: Parallele Aktualisierung der Vertexindizes.

05. Para. Simplifizieren **Verdichtung der Kantenstruktur** (In-Place-Compaction)

Um Performanzeinbußen vorzubeugen werden im letzten Schritt der Simplifizierungsschleife alle ungültigen Kanten aus der Datenstruktur entfernt. Da die anfängliche Sortierung lediglich für die Beseitigung der Kantenreplikate ausgenutzt wurde und für die eigentliche Simplifizierung uninteressant ist, kann an dieser Stelle das Prinzip der *In-Place-Compaction* [DMG10] angewendet werden, bei welcher keine Kopie der Kantendatenstruktur angelegt werden muss.

Für die *In-Place-Compaction* werden zunächst alle aktiven Kanten mit 1 markiert und alle übrigen mit 0. Die entsprechenden Werte werden in einem temporären Feld abgelegt, mit dessen Hilfe im Anschluss eine Präfix-Summen-Berechnung durchgeführt wird. Daraus ergibt sich die Anzahl n der aktuell gültigen Kanten innerhalb der Datenstruktur. Sei N die Anzahl aller in der Datenstruktur vermerkten Kanten, so kann diese in einen

gültigen (0 bis $n-1$) und ungültigen Indexbereich (n bis $N-1$) unterteilt werden. Mittels der Differenz $\Delta n = N - n$ können Rückschlüsse dahingehend gezogen werden, wie viele Elemente maximal aus dem ungültigen in den gültigen Bereich der Kantenstruktur verschoben werden müssen. Dazu müssen die Indexpositionen bzw. Lücken im Intervall $[0, n[$, an welchen sich eine degenerierte Kante befindet, gefunden werden, sodass diese durch eine aktive Kante aus dem Intervall $[n, N[$ ersetzt werden können. Zu diesem Zweck wird die Differenz $\delta_i = i - sum_i$ zwischen den einzelnen Präfix-Teilsummen sum_i und dem zugehörigen Kantenindex i gebildet. Weicht diese Differenz an der Position i von der von $i+1$ ab, so repräsentiert i eine der gesuchten Lücken. Um diese Position im späteren Verlauf wiederzufinden, wird der Wert i an der Position δ_i im Flagarray (oder im temporären Feld) gespeichert. Werden nun die aktiven Kanten e_j aus dem ungültigen Bereich gefiltert, kann über den Wert $\delta_j = sum_j - sum_n$ (Differenz zwischen den Präfix-Summen an den Positionen j und n) die entsprechende Einfügeposition für e_j bestimmt werden, indem an der Position δ_j im Flagarray (bzw. temporären Feld) der zuvor abgelegte Index i einer degenerierten Kante e_i ausgelesen wird. In Abbildung 3.6 wird die Herangehensweise schematisch dargestellt. Bis auf das temporäre Feld für die Berechnung der Präfix-Summe werden für die *In-Place-Compaction* keine zusätzlichen Felder benötigt. Zwischenergebnisse können über das Statusflag der Kanten gehalten werden, da dieses nach der Berechnung der Präfix-Summe nicht mehr benötigt wird.

Elemente	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
Gültig?	J	J	N	J	N	J	J	N	J	J	N	N	J	J	
Flag	1	1	0	1	0	1	1	0	1	1	0	0	1	1	
Präfix-Summe	0	1	2	2	3	3	4	5	5	6	7	7	7	8	9
Differenz	0	0	0	1	1	2	2	2	3	3	3	4	5	5	
Freie Position	2	4	7												
Resultat	0	1	9	3	12	5	6	13	8						

Abbildung 3.6: Funktionsweise der *In-Place-Compaction*.

Da die Möglichkeit besteht, dass während einer *collapse*-Operation zwei Dreiecke aufeinander fallen und dadurch Kantenduplikate entstehen, die die Effizienz des Algorithmus behindern können, wird nach jeder fünften Iteration der Kantenbuffer sortiert und nach Duplikaten durchsucht. Wird ein Replikat gefunden, wird dieses als ungültig eingestuft und durch die nachfolgende *In-Place-Compaction* entfernt. Da der durch das Eliminieren der Replikate erreichte Performanzgewinn (während der Simplifizierung müssen weniger Kanten betrachtet werden) gegenüber der für die Sortierung benötigten Zeit geringer ausfällt, wird der zusätzliche Schritt der Duplikatuntersuchung nur in jeder fünften Iteration vollzogen.

Nach der *In-Place-Compaction* kann der nicht länger benötigte Speicherplatz freigegeben und die neu ermittelte Kantenanzahl als Abbruchkriterium für die Simplifizierungsschleife genutzt werden. Existieren keinerlei Duplikate oder degenerierte Kanten in der Datenstruktur kann die *In-Place-Compaction* übersprungen werden. In Algorithmus 10 wird die geschilderte Vorgehensweise nochmals dargelegt.

```
RADIX SORT edges in parallel by i_max
RADIX SORT edges in parallel by i_min
foreach edge e in parallel do
    e_p = get_previous_edge(e)
    if e_p==e or degenerate(e)
        set_edge_flag(e, 0)
    else set_edge_flag(e, 1)
COMPACT edges in parallel
```

Algorithmus 10: Vorgehensweise zur Verdichtung der Kantendatenstruktur.

Sobald keine *collapse*-Operationen mehr ausgeführt oder die Anzahl der Kanten für ein spezifisches LOD unterschritten wurde, kann das generierte Level gespeichert werden. Um das entsprechende Polygonnetz speichern zu können, müssen zunächst alle irrelevanten Vertizes aus dem *Vertexbuffer* eliminiert werden. Dies wird anhand des zugehörigen Gültigkeitsflags, welches während einer *collapse*-Operation gesetzt wird, vollzogen. Im Anschluss muss der Vorgang hinsichtlich der degenerierten Dreiecke wiederholt werden, da diese auf ungültige Vertizes verweisen. Sind alle relevanten Daten vorhanden, können diese direkt im *Vertex* bzw. *Index Buffer* (VBOs) hinterlegt oder als *Indexed Face Set* (IFS) gespeichert werden.

Befindet sich die Anzahl der Dreiecke unterhalb eines benuterdefinierten Schwellwertes, wird der Simplifizierungsfehler verdoppelt und ein weiteres Level erzeugt. Wurde dieser Schwellwert hingegen überschritten, wurden alle Level produziert und bis auf die generierten VBOs kann der für die Datenstruktur allozierte Speicher freigegeben werden. Abschließend können die VBOs zum Rendern der statischen LODs verwendet werden.

3.4.3 Ergebnisse

Zur näheren Verfahrensanalyse stand ein Testsystem, bestehend aus einem 3.333 GHz Intel Core i7-980X Prozessor, einem 6 GB DDR3-1333 Hauptspeicher und einer NVIDIA GeForce GTX 580 Grafikkarte (841/4204 MHz), zur Verfügung. Für die Implementierung des parallelen Simplifizierungsalgorithmus wurde CUDA verwendet. Die generierten *Indexed Face Sets* wurden mithilfe von OpenGL verarbeitet. Für einen Vergleich hinsichtlich des Ladens von vorberechneten LODs wurde auf eine SATA-II Festplatte (8.5 ms / 64 MB / 7200 rpm) mit einer Lesegeschwindigkeit von ca. 100 MB/s zurückgegriffen. Durch Tabelle 3.3 soll ein Überblick über die während der Tests simplifizierten Modelle wiedergegeben werden. Pro Modell wurde neben der ursprünglichen Dateigröße (IFS), der gesamte Speicherplatz

vermerkt, der für die entsprechend erzeugten LODs benötigt wurde. Für sämtliche Modelle wurde dieselbe Anzahl von Vertexattributen, welche die Koordinaten der Vertexposition und Normalen (k = 6) umfassen, berücksichtigt.

Modell	# Vertizes	# Dreiecke	IFS	LODs
Apache	445,836	807,365	19.4 MB	16.7 MB
St. Dragon	437,645	871,414	19.9 MB	17.5 MB
Buddha	543,652	1,087,716	24.9 MB	24.7 MB
Welsh Dragon	1,105,352	2,210,673	50.5 MB	45.9 MB
Youthful	1,728,305	3,411,563	78.6 MB	70.6 MB
Awakening	2,057,930	4,060,497	93.5 MB	81.4 MB

Tabelle 3.3: Die zur Evaluation verwendeten Modelle, deren maximale Anzahl von Vertizes und Dreiecken sowie die Angaben zum entsprechenden Speicherbedarf.

Die Abbildungen 3.1, 3.3 und 3.7 repräsentieren einige der mithilfe des Verfahrens konstruierten LODs. Ergänzend dazu wurde in Tabelle 3.4 zu jeder generierten Detailstufe die entsprechende Anzahl der zu rendernden Dreiecke aufgelistet. Für die Berechnung des ersten Levels wurde als Richtwert für den Simplifizierungsfehler $\varepsilon = 0.1\%$ von der Diagonalen der ein Modell umschließenden *Bounding Box* angenommen. Mit jeder weiteren Detailstufe wurde dieser Wert verdoppelt. Der Simplifizierungsprozess wurde unterbrochen, sobald ein LOD weniger als 10k Dreiecke beinhaltete. In Abhängigkeit von der Komplexität eines Modells konnten somit zwischen 8 und 10 Level erstellt werden.

Level	Apache	St. Dragon	Buddha	Welsh Dragon	Youthful	Awakening
Original	807,365	871,414	1,087,716	2,210,673	3,411,563	4,060,497
Level 1	321,072	328,733	454,844	871,236	1,255,238	1,514,414
Level 2	187,542	190,002	273,860	469,588	750.257	893,595
Level 3	112,622	110,765	162,282	278,126	455,584	517,640
Level 4	66,433	63,105	92,352	163,848	267.815	284,144
Level 5	39,359	35,218	51,124	95,372	152,008	149,370
Level 6	22,701	19,096	27,504	53,074	83,185	75,750
Level 7	12,908	10,058	14,508	30,768	42,530	37,273
Level 8	7,249	5,132	7,608	17,552	19,368	15,203
Level 9	-	-	-	11,500	8,982	5,639
Level 10	-	-	-	7,476	-	-

Tabelle 3.4: Generierte Detailstufen mit entsprechender Dreiecksanzahl.

Tabelle 3.5 soll den Vergleich zur *QSlim*-Implementierung von Garland und Heckbert [GH97] verdeutlichen. Die Komplexität ihres Verfahrens ist durch die Verwendung einer *Priority Queue* auf $\mathcal{O}(N \log N)$ festgelegt. Im Gegensatz dazu besticht der entwickelte Algorithmus durch eine lineare Komplexität, da der für die Sortierung genutzte *RadixSort* mit einer festen Schlüssellänge auf $\mathcal{O}(N)$ beschränkt ist. Im Vergleich zu den auf die CPU zugeschnittenen Ansätzen wurde eine Beschleunigung um den Faktor 30 bis 40 erzielt.

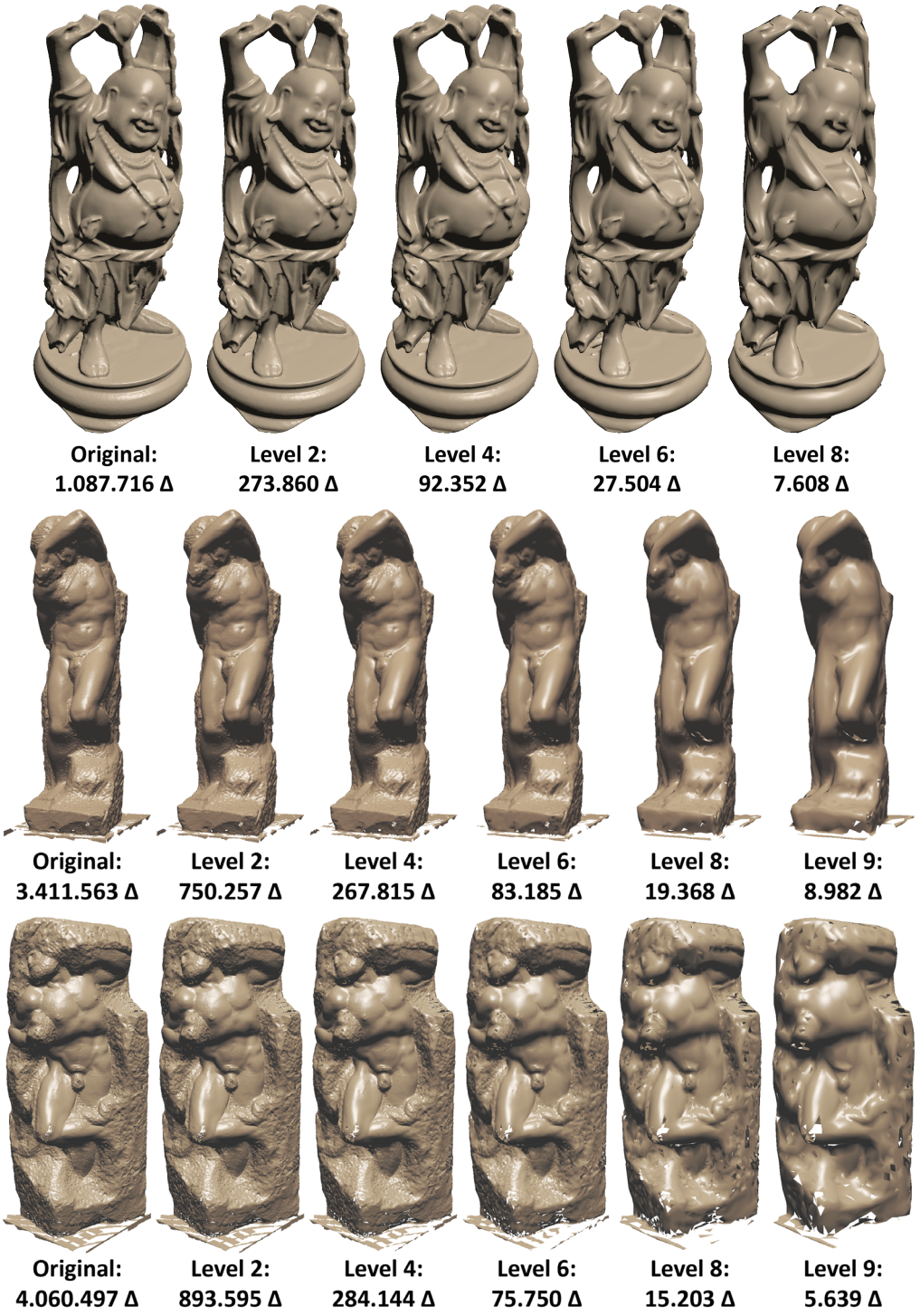


Abbildung 3.7: Einige Ansichten der Modelle (das Original und jedes zweite der entsprechend generierten LODs), die zur Evaluation herangezogen wurden.

Zudem können innerhalb einer Sekunde bis zu 2 Millionen *collapse*-Operationen ausgeführt werden. Die Zeit, welche für die parallele Simplifizierung der Modelle benötigt wird, ist mit der für die Datenübertragung der LODs von der Festplatte zur GPU erforderlichen Transferzeit vergleichbar, jedoch signifikant geringer als die für eine Übertragung über das Netzwerk notwendige Zeitspanne. Der für die parallele Verarbeitung beanspruchte Grafikspeicher ist ungefähr 7 bis 8 mal so hoch wie der für das Originalmodell. Dies entspricht oder liegt gar unterhalb des Speicherverbrauchs für auf der CPU vollzogene Quadrikenberechnungen.

	QSlim		entwickelter Algorithmus			
Modell	Zeit (s)	k Op/s	Speicher	Zeit (s)	k Op/s	Beschleunigung
Apache	8.0	55.7	136.4 MB	0.29	1537	28
St. Dragon	8.0	54.7	139.9 MB	0.28	1564	29
Buddha	10.0	54.4	174.3 MB	0.35	1552	29
Welsh Dragon	22.2	49.8	354.2 MB	0.73	1519	31
Youthful	35.8	48.3	550.7 MB	0.89	1941	40
Awakening	43.9	46.9	655.6 MB	1.03	2003	43

Tabelle 3.5: Vergleich zwischen *QSlim* und dem entwickelten Simplifizierungsverfahren in Bezug auf die benötigte Berechungszeit und den durchführbaren Operationen pro Sekunde.

Gegenüber der von Lindstrom [Lin00] umgesetzten Variante des *Vertex Clustering* wurde eine Leistungssteigerung um den Faktor 10 erreicht. Demzufolge ist der vorgestellte parallele Ansatz um das 30-fache schneller als das *Vertex Clustering* mittels BSP-Bäumen [SG01] bzw. 70 mal so schnell wie jenes unter Verwendung von Octrees [SW03a]. Im Gegensatz zu der von DeCoro und Tatarchuk [DT07] angebotenen parallelen Implementierung des *Vertex Clustering* ist das entwickelte Verfahren jedoch um das 3- bis 4-fache langsamer. Allerdings berücksichtigen DeCoro und Tatarchuk lediglich die Vertexpositionen (k = 3) für ihre Berechnungen. Mit steigender Dimension der Fehlerquadriken relativiert sich diese Diskrepanz, da die meiste Zeit für die Berechnung der optimalen Position des aus einer *collapse*-Operation hervorgehenden Zielvertizes aufgebracht werden muss. In Hinblick auf den qualitativen Unterschied zu den mithilfe von *Vertex Clustering* simplifizierten Modellen benötigte selbst der Octree-Ansatz erheblich mehr Dreiecke, um dieselbe Qualität der generierten Detailstufen zu erreichen.

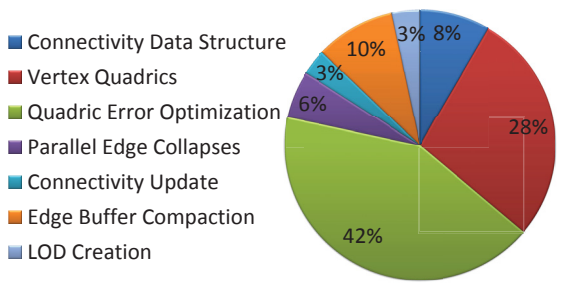


Abbildung 3.8: Relative Laufzeiten für die einzelnen Adaptionsschritte im Vergleich zum Rendering.

Abschließend wurde eine Laufzeitanalyse für die einzelnen parallel ausführbaren Schritte des Adaptions- bzw. Renderingalgorithmus vollzogen (siehe Abbildung 3.8). Bereits bei einer Anzahl von $k = 6$ Vertextattributen wird die meiste Zeit für die Minimierung der Fehlerquadriken benötigt. Da die LDL-Dekomposition eine Komplexität von $\mathcal{O}(N^k)$ aufweist, wird die Laufzeit durch entsprechende Berechnungen dominiert.

3.4.4 Zusammenfassung

In den vorangegangenen Kapiteln wurde die parallele Implementierung eines Simplifizierungsverfahrens diskutiert, welches auf dem von Garland und Heckbert [GH97] entwickelten Ansatz (inklusive einer Betrachtung der metrischen Fehlerquadriken) basiert. Werden alle Kanten kollabiert, die innerhalb ihrer direkten Nachbarschaft über das lokale Minimum verfügen bzw. deren Kosten für die jeweilige Operation minimal sind, entsprechen bzgl. einer vordefinierten Fehlerschranke die generierten Detailstufen (LODs) qualitativ jenen eines sequentiellen Verfahrens. Auf einer handelsüblichen Grafikkarte kann mithilfe der vorgestellten Methode für ein Modell, welches über vier Millionen Dreiecke beinhaltet, eine Menge von LODs in weniger als einer Sekunde erzeugt werden. Dies ist vergleichbar mit der Ladezeit, die benötigt wird um die erzeugten LODs von der CPU auf die GPU zu übertragen und signifikant schneller als ein Netzwerktransfer. Der Vorteil der vorgestellten Methode besteht also darin, dass die LODs aufgrund einer effizienten Datenstruktur parallel auf der Grafikkarte erstellt und somit eine auf der CPU vollzogene Vorverarbeitung der 3D-Modelle irrelevant wird. Des Weiteren wird kein zusätzlicher Festplattenspeicher für die einzelnen Detailstufen benötigt, da diese je nach Bedarf während der Programmlaufzeit bzw. in Echtzeit auf der GPU generiert werden können.

Der wichtigste Nachteil des entwickelten Verfahrens liegt in der sehr zeitintensiven Berechnung des für die Durchführung einer *collapse*-Operation notwendigen Zielvertex. Mit steigender Anzahl von berücksichtigten Vertexattributen dominiert dieser Schritt die absolute Programmlaufzeit. Eine weitere Einschränkung besteht darin, dass während der Simplifizierung eines Modells die Möglichkeit des Kanten flippens nicht überprüft bzw. nicht zur Optimierung des geometrischen Fehlers genutzt wird. Da Letzteres für die verwendeten Modelle unproblematisch ist, könnten bei der Simplifizierung anderer 3D-Objekte sichtbare Artefakte entstehen.

3.5 Multiskalen-Modellierung

Aufgrund der großen Popularität von dreidimensionalen Objekten, werden für interaktive Anwendungen neben den Renderingalgorithmen intuitiv bedienbare Methoden für das Editieren von Modellen benötigt. Um die Rendering- bzw. Darstellungsperformanz der 3D-Modelle zu verbesseren, wird in diesem Kontext oft auf *progressive Meshes* zurückgegriffen, mit welchen die Anzahl der zu rasterisierenden Dreiecke auf ein Minimum reduziert werden kann. Die Datenstruktur umfasst dabei die gröbste Auflösung eines Modells und eine Sequenz von Operationen zur Rekonstruktion der nächst feineren Detailstufen. Im klassischen Fall wird zunächst ein hochwertiges, komplexes 3D-Objekt erzeugt, zu welchem im Anschluss unter Verwendung von Simplifizierungsverfahren das zugehörige *progressive Mesh* generiert wird. Der Nachteil dieses Verfahrens besteht jedoch darin, dass nach jedweden Modellmodifikationen der Simplifizierungsprozess erneut durchlaufen werden

muss. Demzufolge sind zusätzliche Berechnungen vonnöten, aufgrund derer sich die Programmlaufzeit erhöht und die Animation *progessiver Meshes* verhindert wird. Um eine angemessene Lösung bereitzustellen wurde ein echtzeitfähiger, parallel auf der GPU ausführbarer Algorithmus ausgearbeitet [GDG], mit dessen Hilfe die Multiskalen-Modellierung (engl. *multi resolution modeling*) von komplexen *progressiven Meshes* realisiert und deren Animationen verwirklicht werden kann. Durch das entwickelte Verfahren ist es möglich, ein Modell auf einer niedrigen Detailstufe zu deformieren und die vollzogenen Änderungen automatisch zu allen weiteren Objektauflösungen zu propagieren. Für weitreichende Modulationen müssen also lediglich einige Vertizes eines LODs manipuliert werden. Die lokale Geometrie muss zu diesem Zweck in einer effizienten Weise kodiert werden, um die geometrischen Veränderungen korrekt und parallel verarbeiten zu können.

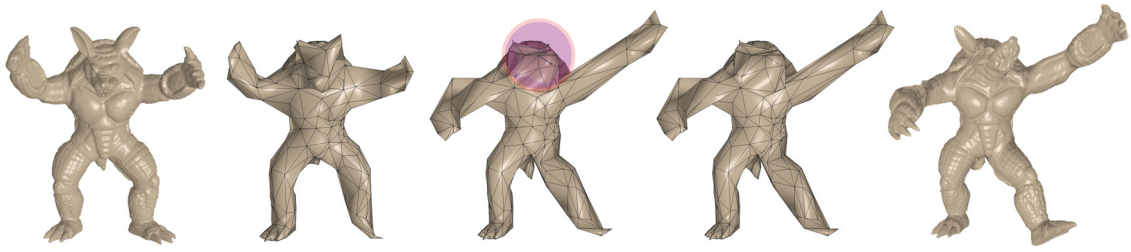


Abbildung 3.9: Multiskalen-Modellierung des zum Armadillo-Modell gehörenden *progressiven Mesh*. Aufgrund der lokalen Kodierung der *split*-Operationen bleiben die geometrischen Details des Originals (links) im modifizierten Modell (rechts) erhalten.

Für den Aufbau der adäquaten Datenstruktur wurde die im vorangegangenen Kapitel diskutierte Methode zur Simplifizierung komplexer 3D-Modelle verwendet. Dieser Ansatz soll im Folgenden als *Basis-Simplifizierer* bezeichnet und für die Modellierung detailreicher Objektstrukturen ausgebaut werden. Für eine derartige Verfahrenserweiterung müssen weitreichende Veränderungen an der Implementierung vorgenommen werden, da sich die während des Editiervorgangs umgesetzten Modifikationen in abgeschwächter Form auch auf die angrenzenden Modellregionen auswirken sollten, um starke Objektverzerrungen zu vermeiden und sanfte Übergange zwischen den einzelnen Passagen einer Oberfläche zu schaffen. Des Weiteren ist es für die Multiskalen-Modellierung erforderlich, dass im Editiermodus zwischen den verschiedenen Detailstufen eines Modells gewechselt und die bereits verrichteten Umgestaltungen entsprechend propagiert werden können (siehe Abbildung 3.9). Letzteres setzt spezifische Anpassungen an der Datenstruktur voraus, vor allem in Anbetracht einer parallelen Datenverarbeitung bzw. einer parallelen Ausführung der *collapse*- und *split*-Operationen (siehe Kapitel 3.1.2). Folglich sollen an dieser Stelle die aus diesem Vorhaben resultierenden Problemstellungen genauer analysiert, deren Lösungsstrategien erläutert und abschließend einige Ergebnisse repräsentiert werden. Zum besseren Verständnis soll jedoch vorab ein Überblick über die grundlegenden Phasen sowie über die wichtigsten Aspekte des entwickelten Verfahrens vermittelt werden.

3.5.1 Grundlagen des Verfahrens

Um die während des Editierens vollzogenen Modell-Modifikationen zwischen den verschiedenen Detailstufen eines 3D-Objektes propagieren zu können, werden *collapse*- und *split*-Operationen (siehe Kapitel 3.1.2) verwendet. Wird eine bestimmte Modellregion umgestaltet, müssen die Positionen der im Einflussbereich (engl. *region of influence* - ROI) der Deformation befindlichen Vertizes korrigiert werden. Diese Anpassungen müssen während einer der durchzuführenden Grundoperationen umgesetzt werden. Demzufolge ist eine geeignete Datenstruktur, welche die relevanten Informationen zur Konnektivität sowie die Daten der Vertexattribute umfasst, ausschlaggebend für die Effizenz des Algorithmus. Die besagte Datenstruktur wird bei diesem Ansatz während der parallelen Simplifizierung eines geladenen 3D-Modells aufgebaut. Die Operationen werden dabei in einer Baumstruktur angeordnet, in welcher vermerkt wird, welche Vertizes miteinander kollabieren dürfen oder zu welchen Vertizes ein spezifischer Punkt aufgespalten wird. Diese Anordnung impliziert eine bestimmte Abfolge, in welcher die einzelnen Operationen durchgeführt werden können. Zusätzlich wird sichergestellt, dass die Nachbarschaftsverhältnisse eines Vertex vor einer *collapse*-Operation mit denen nach dem zugehörigen *Vertex Split* korrespondieren. Letzteres ist besonders wichtig, sollen Fehler oder ungewollte Modellverzerrungen vermieden werden. Des Weiteren können so die getätigten Modifikationen in einfacher Form zu den nächsten Detailstufen weitergeleitet werden. Um die Konnektivität während dieses Vorgangs zu bewahren, muss die lokale Reihenfolge der Operationen über die verschiedenen Teilbäume hinweg beachtet werden. In diesem Sinne darf beispielsweise nur eine Kante kollabieren, wenn keiner der anderen in den anliegenden Dreiecken befindlichen Vertizes in einer der Grundoperationen involviert ist. Für genauere Angaben zu den Ausnahmefällen sei auf Abschnitt 3.5.2.1 verwiesen. Auf Basis des Operationenbaumes kann letztlich ein Modell unter Verwendung der parallelen Grundfunktionen *collapse* und *split* unabhängig von den Detailstufen modelliert und im Anschluss angepasst werden. Im Editiermodus kann auf der aktuell verfügbaren Auflösung ein über einen euklidischen Abstand definierter, beeinflussbarer Modellbereich (ROI) mithilfe eines *Handles* bearbeitet werden. Das *Handle* kann verschoben und rotiert werden. Die Veränderungen wirken sich dabei auf den gesamten ROI aus. Zusammenfassend kann der Algorithmus in zwei Hauptphasen untergliedert werden:

- Simplifizierung des geladenen Modells inklusive der Generierung des zum *progressive Mesh* gehörenden Operationenbaumes
- Multiskalen-Modellierung bzw. Editieren des *progressiven Meshes*.

Vor einer Diskussion beider Teile soll jedoch auf die Kodierung der einzelnen Operationen und auf die Umrechnung zwischen lokalen und globalen Vertexattributen eingegangen werden. Die Umwandlung zwischen lokalen und globalen Attributen ist unter anderem für

die Rekonstruktion der globalen Vertexpositionen relevant, welche aus den bestehenden lokalen Nachbarschaftsverhältnissen gewonnen werden.

3.5.1.1 Die Kodierung der Operationen

Um die Verarbeitung von nicht-mannigfaltigen *Meshes* zu unterstützen, sollten die topologischen Modifikationen nicht als einzelne Komponenten bzw. Operationen in der Datenstruktur hinterlegt, sondern vielmehr im Zusammenhang mit den Dreiecken gespeichert werden. In diesem Kontext sollte auf die im *Basis-Simplifizierer* verwendete Kantenstruktur verzichtet werden, sodass die für das Editieren eines 3D-Modells benötigte Datenmenge auf ein Minimum reduziert werden kann.

Da über jedes Dreieck die zugehörigen Vertizes referenziert werden, können pro Vertex Verweise auf mögliche *collapse*- bzw. *split*-Vertizes in die Datenstruktur aufgenommen werden. Zu diesem Zweck ist es jedoch zwingend erforderlich, dass jedem Vertex eine eindeutige ID zugeordnet wird, die sogenannte *final vertex ID* (FVID). Mithilfe der FVIDs wird eine Ordnung definiert, die sicherstellt, dass lediglich eindeutige Vertexpaare kollabieren können oder ein Vertex zu zwei spezifischen Vertizes aufgespalten wird. Die Vergabe der FVIDs kann allerdings erst nach der erstmaligen Durchführung sämtlicher *collapse*-Operationen erfolgen. Bis dahin müssen auch die Daten zu den einzelnen Operationen gesondert in der Datenstruktur gespeichert werden. Erst wenn die FVIDs vergeben wurden, können die Folgeberechnungen auf die Dreiecksstruktur reduziert werden.

Während einer *collapse*-Operation wird eine Kante e, welche durch die beiden Vertizes v_t und v_u definiert ist, zu einem Vertex v kontrahiert. Dabei verschwindet der Vertex v_u und v_t kann, nachdem die Vertexposition aktualisiert wurde, als Zielvertex v angenommen werden. Zu jeder Operation sollte in der Datenstruktur ergänzend die Anzahl der Vertizes, welche bereits zu dem aktuell aktiven Vertex v kombiniert wurden (im Folgenden als *Vertex Count* bezeichnet), vermerkt werden. Wurde das Modell auf die gröbste Detailstufe (*Base Mesh*) simplifiziert, können die dort enthaltenen Vertizes unter Berücksichtigung des *Vertex Counts* durchnummeriert werden, welcher mit der korrespondierenden *collapse*-Operation hinterlegt wurde. Dementsprechend können die FVIDs zu den Vertizes der nächstfeineren Stufe ermittelt werden, indem an v_t die ID des *collapse*-Vertex v weitergegeben sowie für v_u die Summe aus der ID von v und dem *Vertex Count* von v_t gebildet und als FVID festgelegt wird. Durch diese Vorgehensweise können sämtliche topologischen Veränderungen über die Vertexindzes der Dreiecke kodiert werden. Die Dreiecke können schließlich über die Reihenfolge der durchzuführenden *split*-Operationen, in welcher diese nacheinander generiert werden müssen, gespeichert werden. Die Kodierung der Dreiecke erfolgt über die FVIDs der in der höchsten Detailstufe eines Modells befindlichen Vertizes.

Durch Abbildung 3.10 wird ein einfaches Beispiel zur Nummerierung der Vertizes wiedergegeben. Die Wurzel des Binärbaumes repräsentiert das *Base Mesh*. Da dieses lediglich aus einem Vertex besteht, erhält dieser die FVID mit dem Wert 0. Der Vertex v

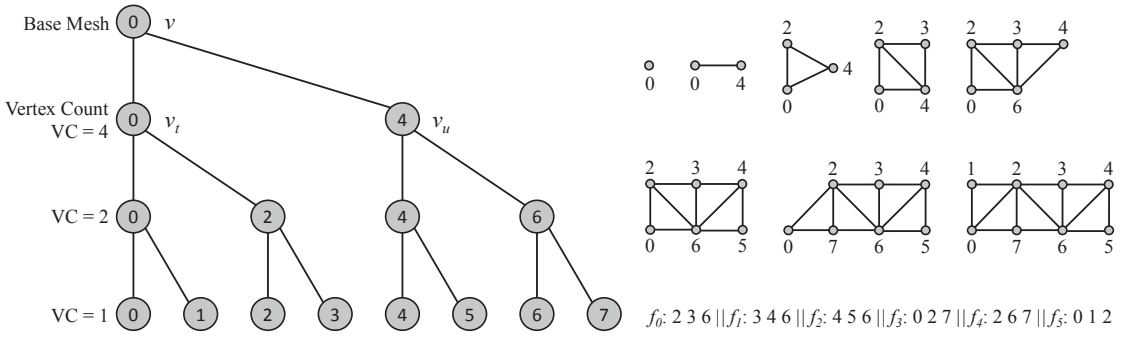


Abbildung 3.10: Schematische Darstellung zur Berechnung der *final vertex IDs* und der Kodierung der zu generierenden Dreiecke.

entspricht dem Zielvertex einer *collapse*-Operation, bei welcher v_t und v_u zusammengefasst wurden. Aufgrund der Tatsache, dass v_t als *collapse*-Vertex v recycelt wurde, erhält dieser die gleiche FVID wie v. Für v_u wird der Wert des *Vertex Counts* von v_t hinzuaddiert. Sollen im späteren Verlauf die Vertizes v_t und v_u wieder zu v kollabiert werden, wird zu v_t der Vertex mit der nächst höheren FVID gewählt. Ist das Vertexpaar gefunden, kann im Anschluss die *collapse*-Operation durchgeführt werden.

Die resultierende Kodierung der Dreiecke in Verbindung mit der Abfolge der *split*-Operationen wird in der rechten Hälfte der Abbildung 3.10 verdeutlicht. Zu Beginn soll das Dreieck f_0 generiert werden, welches durch die FVIDs der zugehörigen Vertizes definiert ist. Ist einer der Dreiecksvertizes mit der entsprechenden FVID noch nicht vorhanden, muss der Vertex mit der nächst kleineren FVID gesplittet werden. Dies wird solange wiederholt, bis alle Dreiecke des ursprünglichen Modells rekonstruiert wurden.

Zusammenfassend kann also gesagt werden, dass nach der Generierung des *Base Meshes* sämtliche *collapse*- und *split*-Operationen über die FVIDs der Vertizes, inklusive aller zugehörigen Vertexattribute, den Verfeinerungskriterien, den zu generierenden Dreiecken (unter Berücksichtigung möglicher Modifikationen) sowie die Operationen der Zwischensequenzen, kodiert bzw. abgerufen werden können. Die Daten werden dabei nicht weiter komprimiert, um das Editieren des Modells zu ermöglichen. Die lokale Ordnung wird zusätzlich gewährleistet, indem neben dem maximalen Simplifizierungsfehler der Fehler für die einzelnen Nachbarvertizes (plus einer kleinen Abweichung ε) gespeichert wird. Demnach müssen lediglich die Fehler der *collapse*- bzw. *split*-Operationen überprüft werden, um deren strikte lokale Reihenfolge sicherstellen zu können.

3.5.1.2 Lokale und Globale Vertexattribute

In Komprimierungsverfahren werden die *Offsets* von Vertizes meistens über das lokale Koordinatensystem des entsprechenden *split*-Vertex kodiert. Dies führt zu einer Verbesserung der Kompressionsraten bzw. zur einer Reduktion der Datenmengen, im vorliegenden Kontext können jedoch zusätzlich die auf einen *split*-Vertex wirkenden Transformationen direkt

an dessen komplette Nachfolgerschaft vererbt werden. Um eine gleichmäßige Weiterleitung der Modifikationen an benachbarte *split*-Vertizes zu gewährleisten, beruht das lokale Koordinatensystem auf dem aus allen angrenzenden Vertizes ermittelten Durchschnitt (siehe Abbildung 3.11). In Bezug auf die Grundoperationen *collapse* und *split* ist die lokale Position, die Normale sowie die zugehörige Tangente (P_{int}, N_{int}, T_{int}) als gewichteter Durchschnitt aus v und allen an v_t bzw. v_u angrenzenden Vertizes definiert. Die *Offsets* zur globalen Position bzw. Normalen (P_{off}, N_{off}) können anschließend mithilfe des lokalen Koordinatensystems, welches durch N_{int} und T_{int} aufgespannt wird, kodiert werden. Da für die Tangente lediglich ein Freiheitsgrad existiert, kann diese als Rotation um die Normale N verwaltet werden. Die genaue Berechnung der Daten wird, inklusive entsprechender Formeln, in Verbindung mit der Implementierung im Folgekapitel 3.5.3.1 besprochen.

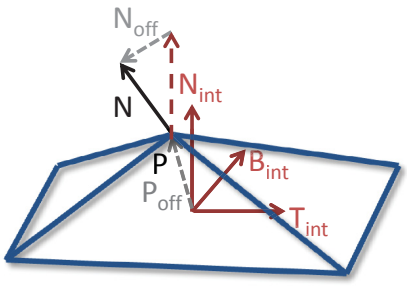


Abbildung 3.11: Die Kodierung der Vertexattribute relativ zum aus der Nachbarschaft interpolierten, lokalen Koordinatenssystem.

3.5.2 Generierung des *progressiven Meshes*

Wie bereits erwähnt wird der bestehende *Basis-Simplifizierer* [GDG11] zur Generierung des zu einem Modell gehörenden *progessiven Meshes* sowie zur Konstruktion der für die Multiskalen-Modellierung benötigten Datenstruktur verwertet. Da der *Basis-Simplifizierer* auf die Erzeugung von *statischen* LODs beschränkt ist, muss dieser erweitert bzw. teilweise abgeändert werden, sodass sowohl die Echtzeit-Modellierung als auch die entsprechend parallele Adaption am *progressiven Mesh* vollzogen werden kann. Die notwendigen Erweiterungen sowie Anpassungen werden in Abbildung 3.12 dargelegt.

Um zwischen den verschiedenen Detailstufen des Modells wechseln bzw. die vorgenommenen Objektdeformationen zu den nächstfeineren Auflösungen propagieren zu können, müssen zusätzlich die für die *split*-Operationen notwendigen Informationen gesammelt werden. In diesem Kontext wird eine identische Nachbarschaft vorausgesetzt, welche zu jeder Zeit vor der Ausführung einer *collapse*-Operation col_v und nach dem korrespondierenden *Vertex Split* spl_v gewährleistet sein muss. Die einheitlichen Nachbarschaftsverhältnisse sind für die Berechnung der lokalen und globalen Vertexattribute maßgeblich. Stimmen die Nachbarschaftsbeziehungen für die Operationen col_v und spl_v nicht überein, ist die Interpolation der Vertexpositionen für $\overline{v}$ bzw. v_t und v_u fehlerhaft, was sich auf die gesamte parallele Adaption auswirkt und in ungewollten Objektverzerrungen resultiert. Als Begleiterscheinung muss zusätzlich eine strikte Reihenfolge bei der Durchführung der Operationen eingehalten werden.

Im Folgenden sollen nun weitere Problemstellungen aufgezeigt und im Anschluss die daraus resultierenden algorithmischen Veränderungen erörtert werden.

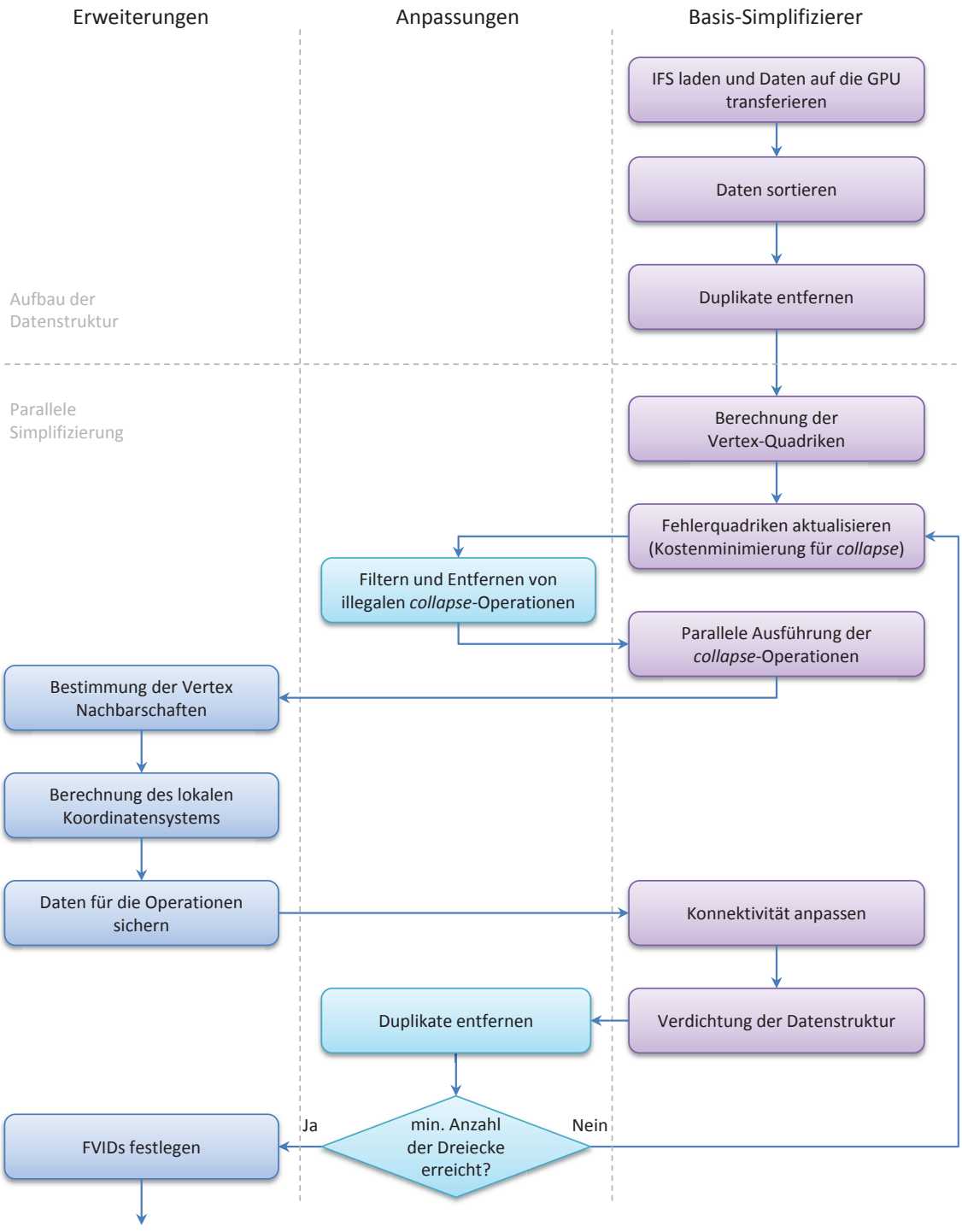


Abbildung 3.12: Schema zur parallelen Generierung des *progressiven Meshes* einschließlich der vorgenommenen Erweiterungen (links) und Anpassungen (Mitte) des zuvor entwickelten *Basis-Simplifizierers* (rechts).

3.5.2.1 Problemstellungen

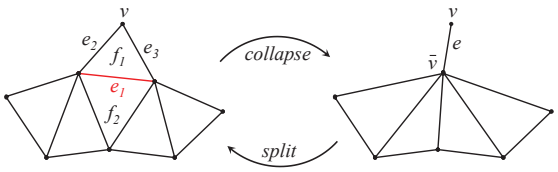
Beim *Basis-Simplifizierer* werden, entsprechend den Algorithmen 7 und 8, pro Kante e zunächst die Kosten für die zugehörige *collapse*-Operation ermittelt und im Anschluss überprüft, ob diese dem lokalen Minimum entsprechen. Ist dies der Fall, wird die jeweilige Operation durchgeführt und e zu einem Vertex $\overline{v}$ kontrahiert. Da außer der Minimalitätsanalyse keine weiteren Bedingungen an die Ausführung einer *collapse*-Operation geknüpft sind, ergeben sich im Bezug auf den Editiermodus sowie für das Propagieren der Modifikationen folgende Problemstellungen:

Situation 01: Seien neben der Kante e_1, welche kollabiert werden soll, e_2 bzw. e_3 zwei Randkanten innerhalb eines Dreiecks (siehe Abbildung 3.13a). Angenommen, die *collapse*-Operation für e_1 wird ausgeführt, dann verschmelzen e_2 bzw. e_3 zu einer Kante e, welche im Nachhinein an keinem Dreieck mehr angrenzt. Während des inversen *split*-Vorgangs werden die degenerierten Dreiecke f_1 bzw. f_2 wieder erzeugt und über die an $\overline{v}$ angrenzenden Dreiecke in die Nachbarschaft integriert. Da der Vertex v jedoch nicht erreicht wird, kann keine Verbindung zu f_1 hergestellt werden. Die Konnektivität zwischen den Elementen der Datenstruktur geht verloren. Eine derartige *collapse*-Operation muss somit verhindert werden.

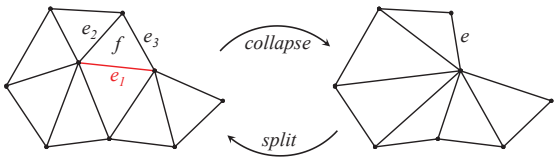
Situation 02: Sei e_3 eine Randkante und einer der beiden anderen Dreieckskanten e_1 oder e_2 soll kollabiert werden (siehe Abbildung 3.13b). In diesem Fall wird das Dreieck f auf die Kante e reduziert, welche abschließend als Randkante markiert werden muss. Da die Dreiecksvertizes von f vor dem Kollabieren mit weiteren in der Nachbarschft befindlichen Dreiecken verbunden sind, darf die *collapse*-Operation durchgeführt werden. Lediglich das Randkantenflag der resultierenden Kante e muss gesetzt werden. Wird hingegen die Randkante e_3 kollabiert, muss keine Statusänderung erfolgen.

Situation 03: Seien e_1 und e_2 zwei Kanten, deren *collapse*-Kosten dem lokalen Minimum entsprechen (siehe Abbildung 3.13c). Zunächst scheinen die jeweiligen *collapse*-Operation unproblematisch zu sein. Im *Basis-Simplifizierer* werden diese auch durchgeführt. Da im Zuge des Editierens die Änderungen am *Mesh* zu den nächst feineren Detailstufen propagiert werden, müssen die zugehörigen *split*-Operationen vollzogen werden. Da die beiden Vertizes der Kante e_3 jedoch nicht gleichzeitig gesplittet werden können, müssen diese nacheinander absolviert werden. Werden die *split*-Operationen der Kante e_3 nacheinander ausgeführt, entspricht die Nachbarschaft wiederum nicht derjenigen, die vor der *collapse*-Operation bestand. Die Interpolation der Vertexpositionen würde demnach falsche Ergebnisse liefern, wodurch das Original nicht mehr rekonstruiert werden könnte. Das Kollabieren der Kanten e_1 und e_2 darf also nicht gleichzeitig, sondern muss nacheinander erfolgen, sodass identische Nachbarschaftsverhältnisse garantiert werden können.

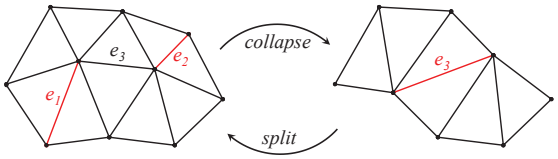
Situation 04: Seien e_1 und e_2 zwei Kanten einer im Modell vorhandenen Dreieckspyramide, die kollabiert werden sollen (siehe Abbildung 3.13d). Wird nur eine der beiden Kanten, z.B. e_1, zu einem Vertex kontrahiert, fallen zwei der vier Dreiecke aufeinander, wodurch Duplikate entstehen. Für den korrekten Ablauf des Verfahrens sind Replikate zunächst vollkommen irrelevant. Wird im Anschluss jedoch e_2 kollabiert (die Duplikate verschwinden zur selben Zeit) oder werden beide *collapse*-Operationen gleichzeitig durchgeführt, werden bei den zugehörigen *splits* die Dreiecksduplikate nicht repliziert, wodurch im resultierenden *Mesh* Fehler auftreten. Da dieser Fall auf Situation 03 zurückgeführt werden kann, müssen bei der Implementierung keine speziellen Abfragen integriert werden. Allerdings muss pro Operation die Anzahl der degenerierten Dreiecke festgehalten werden, sodass diese im späteren Verlauf generiert werden können.



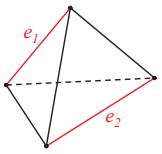
(a) Situation 01



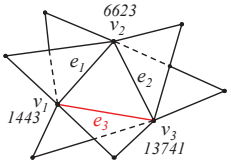
(b) Situation 02



(c) Situation 03



(d) Situation 04



(e) Situation 05

Abbildung 3.13: Schematische Darstellung der einzelnen Problemfälle.

Situation 05: Es soll ein dreidimensionales Gebilde (in Abbildung 3.13e dargestellt) simplifiziert werden, dessen innen liegendes Loch während des Prozesses geschlossen wird. Die Kanten e_1, e_2 und e_3 stellen in diesem Kontext keine Randkanten dar. Wird nun e_3 kollabiert, fallen e_1 und e_2 aufeinander. Die Kantenduplikate werden jedoch nicht erkannt, da durch e_1, e_2 und e_3 kein Dreieck definiert wird, über welches die zugehörigen Gültigkeitsflags der Kanten gesetzt werden können. Zudem wurde für die Kantenstruktur eine Sortierordnung festgelegt, welche durch die Ausführung der zu e_3 gehörenden *collapse*-Operation verletzt werden könnte. Bezüglich des in Abbildung 3.13e angegebenen Beispiels wird v_3 letztlich auf den *collapse*-Vertex v_1 gelinkt, wodurch dessen ID von 13741 auf 1443 gesetzt und dadurch die Indexreihenfolge der Kante e_2 umgekehrt wird. Die in der Datenstruktur enthaltenen Kantenduplikate führen in Verbindung mit der möglichen Indexverschiebung zu Fehlinterpretationen bei der Nachbarschaftssuche und damit zu Interpolationsfehlern. Diese müssen gefiltert und entfernt werden.

3.5.2.2 Vorzunehmende Modifikationen

Zu Beginn der Simplifizierung wird das *Indexed Face Set* (IFS) eines Modells geladen. Die Vertexattribute und Dreiecksindizes werden auf die GPU transferiert und in einem *Vertex* bzw. *Index Buffer* (VBOs) gespeichert. Im Anschluss wird wie im *Basis-Simplifizierer* die Kantenstruktur aufgebaut. Während dieser Prozedur werden die Randkanten markiert und alle Kantenduplikate entfernt. Um im späteren Verlauf eine fixe Nachbarschaft für die Operationen garantieren zu können, werden abweichend vom Originalalgorithmus für jedes Dreieck die Kantenindizes vermerkt.

Nun kann mit der eigentlichen parallelen Simplifizierung fortgefahren werden. Dazu werden zunächst die Vertexquadriken berechnet, um die Kosten der *collapse*-Operationen bewerten und die optimale Position des entsprechenden Zielvertex $\overline{v}$ zu ermitteln. Um Speicherplatz zu sparen, könnte dieser Schritt in die Simplifizierungsschleife integriert werden, da diese nicht mehr benötigt werden, sobald die Anzahl der legalen *collapse*-Operationen feststeht. Wurden alle Quadriken über die an einem Vertex anliegenden Dreiecke akkumuliert und die der Randkanten angepasst, kann die Optimierung der Fehlerquadriken vollzogen werden, in deren Verlauf überprüft wird, ob die Kosten eines *collapse*-Kandidaten dem lokalen Minimum entsprechen. Für die Gewährleistung einer strikten Reihenfolge der Operationen wird für den Simplifizierungsfehler ε_v einer Operation col_v jedoch folgendes angenommen:

$$\varepsilon_v = \max\left(\varepsilon_{quadric},\ (1+\varepsilon_{float}) \max_{i \in N(v)} \varepsilon_i^s\right), \tag{3.14}$$

wobei $\varepsilon_{quadric}$ der durch die Quadrik des Zielvertex v implizierten Abweichung, ε_i^s dem zuvor akzeptierten Simplifizierungsfehler eines aus der Nachbarschaft $N(v)$ von v stammenden Vertex i und $(1+\varepsilon_{float})$ einem Multiplikator entspricht, durch welchen sichergestellt wird, dass der für col_v gespeicherte Fehler größer ausfällt, als jener der zuletzt in der direkten Nachbarschaft $N(v)$ ausgeführten *collapse*-Operation col_i. Der pro *collapse*-Operation ermittelte maximale Fehler wird in der Datenstruktur hinterlegt. Mit diesem Wert kann später bei der Modellierung bzw. im Adaptionsvorgang die korrekte Abfolge der *collapse*- bzw. *split*-Operationen garantiert werden, indem beispielsweise erst ein *Vertex Split* spl_v durchgeführt wird, wenn in seiner direkten Nachbarschaft kein Vertex mit einem höheren Fehlerwert existiert. Wurde für alle in der Nachbarschaft befindlichen Vertizes ein geringerer Fehlerwert vermerkt, als der durch die zu spl_v inverse Operation col_v verursachte Fehler, kann davon ausgegangen werden, dass die Nachbarschaftsverhältnisse nach der Ausführung von spl_v, mit denen vor der Operation col_v identisch sind. Umgekehrt darf während der Adaption kein Vertexpaar kontrahiert werden, wenn in der Nachbarschaft *collapse*-Operationen mit einem niedrigeren maximalen Fehlerwert vorhanden sind.

Wie im ursprünglichen Algorithmus werden für die Vertizes einer Kante in der Datenstruktur Verweise auf den *collapse*-Kandidat hinterlegt, dessen Fehler dem lokalen

Minimum entspricht. Vor der Ausführung der Operationen müssen jedoch gemäß der Problemsituationen 01, 03 und 04 aus Abschnitt 3.5.2.1 unerlaubte Kantenkontraktionen verhindert werden. Demnach darf innerhalb eines Dreiecks keine Kante kollabiert werden, wenn es sich bei den beiden anderen um Randkanten handelt oder wenn die Vertizes einer im Dreieck befindlichen Kante in verschiedenen *collapse*-Operationen involviert sind oder wenn die Vertizes einer Kante, die nicht als *collapse*-Kandidat markiert wurde, kollabiert werden sollen. Bei Letzterem darf lediglich die angrenzende Kante mit den niedrigeren Kosten kollabieren. Die andere Operation kann anschließend in einer weiteren Iteration durchgeführt werden.

01.
Modifikationen
Illegale *collapse*-Operationen entfernen

Um illegale *collapse*-Operationen zu filtern wird zunächst die Randkantenproblematik analysiert. Dazu wird für jede Kante die Anzahl der angrenzenden Dreiecke bestimmt, die sich nach der Ausführung der *collapse*-Operationen ergeben würde. Für die parallele Umsetzung werden alle aktiven Dreiecke betrachtet, deren Kanten nicht kontrahiert werden sollen und im Zuge dessen auch nicht verschwinden. Hierfür werden die anfänglich pro Dreieck gesicherten Referenzen auf die zugehörigen Kanten benötigt. Mithilfe der Referenzen kann eine angemessene Abfrage vollzogen und die jeweilige Dreiecksanzahl um 1 inkrementiert werden. Existieren im Nachhinein Dreiecke, deren Kanten anschließend über keine weitere Verbindung zu einem Dreieck verfügen, werden die entsprechenden Operationsflags zurückgesetzt. Mithilfe dieser Herangehensweise kann die in Situation 02 (siehe Abschnitt 3.5.2.1) beschriebene Statusänderung bzw. das zustandsabhängige Setzen des Randkantenflags während der Ausführung einer *collapse*-Operation vernachlässigt und durch den Verzicht auf dieses Flag der Speicherverbrauch reduziert werden. Für die zu Beginn durchgeführte Anpassung der Vertexquadriken genügt demnach die Verwendung eines temporären Feldes.

Um zu erfahren ob sich innerhalb eines Dreiecks zwei in verschiedene *collapse*-Operationen involvierte Vertizes befinden, deren gemeinsame Kante jedoch nicht kollabiert werden soll, muss zunächst der Status aller Dreiecksvertizes geklärt werden. Zu diesem Zweck wird ein temporäres Feld angelegt und in einem parallelen Schritt über alle aktiven Kanten gelaufen. Soll eine Kante e kollabiert werden, wird im temporären Feld für die zugehörigen Vertizes der Index von e hinterlegt, anderenfalls der Wert -1. Werden nun die Kanten aller aktiven Dreiecke betrachtet und die zu den Vertizes gespeicherten Indizes stimmen nicht mit dem Index der jeweiligen Kante überein, kann mithilfe der gesicherten Indizes direkt auf das Statusflag der angrenzenden Kanten zugegriffen werden. Die Kante mit den maximalen Kosten darf nicht kollabiert werden, um die eindeutige Nachbarschaft sowie die strikte Reihenfolge der Operationen garantieren zu können.

02.
Modifikationen
Sicherung relevanter Operationen

Nachdem alle gültigen *collapse*-Operationen gefiltert wurden, können diese ausgeführt werden. Währenddessen müssen die für die Umsetzung der *split*-Operationen notwendigen Daten erfasst werden. Hierfür wird zu allen an einer *collapse*-Operation beteiligten Ver-

tizes die Nachbarschaft bestimmt, indem alle aktiven Dreiecke durchlaufen, die Anzahl der angrenzenden Vertizes ermittelt sowie deren Attribute aufsummiert und die daraus hervorgehenden Informationen in der Datenstruktur gespeichert werden. Diese werden für die Interpolation des lokalen Koordinatensystems bzw. für die Transformation der globalen Vertexattribute in lokale Merkmale und die damit im Zusammenhang stehende Kodierung der resultierenden *Offsets* verwendet.

Des Weiteren wird, als Konsequenz aus der Problemsituation 04 (siehe Abschnitt 3.5.2.1), pro *collapse*-Operation die Anzahl der degenerierten Dreiecke bestimmt. Dieser Wert wird jedoch nicht in der Datenstruktur hinterlegt. Aufgrund der vorausgesetzten Ordnung der Operationen verschwinden auch die im Modell enthaltenen Dreiecke in einer bestimmten Reihenfolge und müssen dementsprechend auch wieder erzeugt werden. Dadurch wird für jede Operation col_v lediglich eine Referenz auf das erste degenerierte Dreieck verbucht, welches durch col_v verschwunden ist. Bei einem späteren *Vertex Split* müssen somit alle Dreiecke rekonstruiert werden, die sich zwischen den durch zwei aufeinanderfolgende *collapse*-Operationen referenzierten Dreiecken befinden.

Sind alle in einer Iteration der Simplifizierungsschleife möglichen *collapse*-Operationen absolviert worden, wird wie beim *Basis-Simplifizierer* die Konnektivität zwischen den einzelnen Komponenten der Datenstruktur aktualisiert und die Kantenstruktur mittels einer *In-Place-Compaction* verdichtet. Hierbei muss lediglich darauf geachtet werden, dass die pro Dreieck integrierten Verweise auf deren Kanten ebenfalls angepasst werden. Um das in Abschnitt 3.5.2.1 geschilderte fünfte Problem zu lösen, müssen zusätzlich noch alle Kantenduplikate entfernt werden. Die Duplikatsuche entspricht der am Anfang der Simplifizierung durchgeführten Replikatentfernung. Beim *Basis-Simplifizierer* wurde dieser Schritt nur sporadisch durchgeführt, um eine Reduktion des Speicherverbrauchs herbeizuführen. In diesem Kontext muss dies jedoch bei jeder Iteration geschehen, um die Eindeutigkeit der Operationen gewährleisten und mögliche Fehlinterpretationen ausschließen zu können.

03.
Modifikationen
Vergabe der FVIDs

Für den *Basis-Simplifizierer* wurde als Abbruchkriterium pro LOD ein ε angegeben. Da für die nachfolgende Modellierungsphase lediglich die gröbste Detailstufe von Interesse ist, wurde eine benutzerdefinierte minimale Anzahl von Dreiecken bzw. eine maximale Anzahl von Iterationen festgelegt, die nicht unter- bzw. überschritten werden sollte. Wird dieser Schwellwert erreicht, können alle relevanten Daten für die parallele Adaption zusammengefasst, über die Vergabe der *final vertex IDs* (FVIDs) kodiert und in der korrekten Reihenfolge gespeichert werden. Die genaue Vorgehensweise wurde bereits in Abschnitt 3.5.1.1 erläutert.

Ein komplettes Listing zu den Elementen der Datenstruktur, welche zur Generierung des *progessiven Meshes* und zur Konstruktion des Operationenbaumes benötigt werden, kann in Tabelle 3.6 eingesehen werden. Dabei sollte berücksichtigt werden, dass während

buffers	elements	bytes per entry
faces	index VBO	12
	active flag	4
	edge index (x3)	12
edges	vertex index (x2)	8
	active flag	4
	collapse state	4
	optimal placement	$4k$
	temporary data	28
vertices	vertex VBO	$4k$
	vertex quadric	$2k^2 + 6k + 4$
	min edge ID	4
	min edge cost	4
	active flag	4
	adjacent vertex attributes	$4k$
	adjacent vertex count	4
	operation index	4
	split index	4
	FVID (x2)	8
operations	vertex local attributes (x2)	$8(k + 1)$
	vertex index (x2)	8
	target vertex	4
	cost	4
	split index (x2)	8
	child count	4
	face index	4
	temporary	4
temporary	edge sort order (x2)	8
	edge sort key	4
	edge prefix sum (scan)	4

Tabelle 3.6: Elemente der für die *Mesh*-Simplifizierung verwendeten Datenstruktur, wobei k der Anzahl an berücksichtigten Vertexattributen entspricht.

der Simplifizierung nicht alle Komponenten beansprucht und bestimmte Felder im Laufe des Verfahrens freigegeben werden können.

3.5.3 Der Editiermodus

Um das Verständnis für die folgenden Verfahrensaspekte zu fördern, sollen an dieser Stelle die aus der Simplifizierung resultierenden Begebenheiten nochmal zusammengefasst werden. Während der Simplifizierung wurde ein *progressives Mesh* erzeugt. Hierbei handelt es sich um ein Polygonnetz, welches die gröbste Detailstufe eines dreidimensionalen Modells darstellt. Dieses Polygonnetz wird auch als *Base Mesh* bezeichnet. Die darin enthaltenen Vertizes werden durch globale Attribute (Position, Normale usw.) repräsentiert. Um ausgehend vom *Base Mesh* die feineren Detailstufen eines Modells rekonstruieren zu können, wurden zusätzlich zu den global bestehenden Grundverhält-

nissen die relativen Abweichungen bzw. lokalen Differenzen zwischen den nächsthöheren Auflösungen in der Datenstruktur hinterlegt. Zur Berechnung der lokalen Differenzen oder *Offsets* wurden während einer *collapse*-Operation col_v, bei welcher ein Vertexpaar v_t und v_u zum Zielvertex v kollabierte, die Attribute der in der direkten Nachbarschaft von v_t bzw. v_u befindlichen Vertizes aufsummiert und durch deren Anzahl dividiert. Der daraus resultierende Durchschnitt wurde als Referenzpunkt für das lokale Koordinatensystem zu v_t und v_u angesehen. Die *Offsets* können also als relative Abweichungen oder lokale Differenzen zwischen den Attributen von v_t bzw. v_u und den entsprechend ermittelten Durchschnittswerten aufgefasst werden. Wird nun beim Editieren die zu col_v inverse *split*-Operation spl_v ausgeführt, wird der Vertex v in die beiden Vertizes v_t und v_u aufgespalten. Diese werden in das bestehende Polygonnetz aufgenommen und mit den Nachbarvertizes von v verbunden (siehe Abbildung 3.14). Um die globalen Attribute für v_t und v_u zu reproduzieren, werden wiederum die Attribute der Nachbarn aufsummiert und der jeweilige Durchschnitt gebildet. Zu dem Durchschnitt werden anschließend die zuvor gesicherten *Offsets* von v_t bzw. v_u hinzuaddiert.

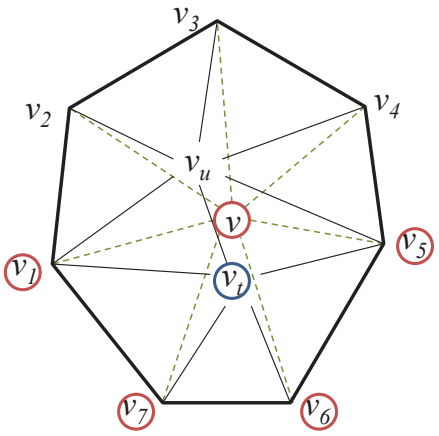


Abbildung 3.14: Die in der *split*-Nachbarschaft von v_t befindlichen Vertizes (rot markiert) werden zur Interpolation des Referenz-Koordinatensystems von v_t verwendet.

Zur Vermeidung von möglichen Fehlern muss die Nachbarschaft eines an einer *collapse*-Operation beteiligten Vertex vor deren Ausführung mit der nach dem korrespondierenden *Vertex Split* übereinstimmen. Dies wird durch die strikte Abfolge der durchzuführenden Operationen gewährleistet. Da durch die definierte Reihenfolge genau festgelegt ist (siehe Abschnitt 3.5.1.1), welche Vertizes bei einer *collapse*-Operation zu einem Vertex verschmelzen, muss in der Datenstruktur pro involviertem Vertex nur eine Referenz auf den Zielvertex gespeichert werden. Für die *split*-Operation müssen hingegen die beiden Indizes auf die zu generierenden Vertizes vermerkt werden. Da diese in der aktuellen Detailstufe noch nicht aktiv sind, können an den entsprechenden Indexpositionen lediglich die lokalen *Offsets* abgerufen werden.

Wird das *progressive Mesh* editiert, werden entweder die globalen Attribute der im *Base Mesh* enthaltenen Vertizes direkt oder die durch die *split*-Operationen referenzierten bzw. kodierten *Offsets* verändert. Die Modelländerungen werden dabei immer anhand der globalen Attribute realisiert und müssen im Anschluss auf entsprechend modifizierte lokale Attribute abgebildet werden. Des Weiteren müssen die editierten Daten sowohl zu den nächsthöheren als auch zu den nächstgröberen Detailstufen propagiert werden, um die vollzogenen Deformationen beizubehalten. Demzufolge ist eine ständige Konvertierung zwischen lokalen und globalen Vertexattributen erforderlich.

3.5.3.1 Umrechnung zwischen lokalen und globalen Attributen

Um während dem zu einer *collapse*-Operation col_v inversen *Vertex Split* spl_v Rückschlüsse auf die globalen Vertexattribute von v_t und v_u ziehen zu können, werden die lokalen Differenzen bzw. *Offsets* benötigt. Werden in der Datenstruktur lediglich die regionalen Abweichungen zum *Base Mesh* gespeichert, hat dies beispielsweise den Vorteil, dass ein Modell als Ganzes verschoben oder rotiert werden kann. Eine Translation oder Rotation wirkt sich nicht auf die lokal bestehenden Beziehungen aus, sodass ausschließlich die globalen Attribute der *Base Mesh*-Vertizes angepasst werden müssen. Würden hingegen die globalen Attribute pro Vertex gesichert, müsste der komplette Datensatz aktualisiert werden.

Da bei der Durchführung von col_v für v_t und v_u lediglich die direkten Vertexnachbarn inklusive v verfügbar sind, es jedoch eines Referenz-Koordinatensystems für die Berechnung der *Offsets* bedarf, beschränkt sich die Interpolation des lokalen Koordinatensystems eben auf diese Nachbarvertizes. Um Interpolationsfehler zu vermeiden muss zu jeder Zeit die identische Nachbarschaft für col_v bzw. spl_v garantiert werden. Die eindeutigen Nachbarschaftsverhältnisse sind vor der Ausführung von col_v bzw. nach dem *Vertex Split* spl_v gegeben. In Abbildung 3.14 wurde beispielhaft die für v_t referenzierte Nachbarschaft gekennzeichnet. Da es sich bei v um den Zielvertex von col_v handelt, fließt dieser mit einer doppelten Gewichtung in die Berechnungen ein.

Sind die *Offsets* der Attribute aus dem globalen Koordinatensystem und das aus der Nachbarschaft interpolierte lokale Koordinatensystem gegeben, so lassen sich die lokalen *Offsets* (P_{local}, N_{local} and α) folgendermaßen bestimmen:

$$P_{local} = (P_{off} \cdot T_{int}, P_{off} \cdot B_{int}, P_{off} \cdot N_{int})^t \tag{3.15}$$

$$N_{local} = (N_{off} \cdot T_{int}, N_{off} \cdot B_{int}, N_{off} \cdot N_{int})^t \tag{3.16}$$

$$T_{ortho} = normalize\,(T_{int} - N\,(N \cdot T_{int})) \tag{3.17}$$

$$\alpha = \arctan \frac{T_{ortho} \cdot (N \times T)}{T_{ortho} \cdot T}, \tag{3.18}$$

wobei P, N und T der Position, Normalen und Tangenten von v_t oder v_u entsprechen. Die interpolierte Bitangente ist durch $B_{int} = N_{int} \times T_{int}$ definiert und α steht für eine Rotation von T um die Normale des lokalen Koordinatensystems. Die relevanten *Offsets* für die Position und der Normalen lauten:

$$P_{off} = P - P_{int} \tag{3.19}$$

$$N_{off} = N - N_{int}. \tag{3.20}$$

Für die globalen Komponenten gelten folgende Zusammenhänge:

$$P_{off} = (T_{int} \quad B_{int} \quad N_{int})^t P_{local} \tag{3.21}$$

$$N_{off} = (T_{int} \quad B_{int} \quad N_{int})^t N_{local} \tag{3.22}$$

$$P_{global} = P + P_{off} \tag{3.23}$$

$$N_{global} = N + N_{off} \tag{3.24}$$

$$T_{global} = T_{int}\cos(\alpha) + (N_{local} \times T_{int})\sin(\alpha). \tag{3.25}$$

3.5.3.2 Verbreitung editierter Modifikationen

Nach dem Editieren müssen die vollzogenen Änderungen zu den verschiedenen Detailstufen des *Meshes* propagiert werden. Für die nächsthöheren Modellauflösungen werden die Modifikationen automatisch über die *split*-Operationen weitergetragen, indem die aus dem Editiervorgang hervorgehenden *Offsets* mit den zuvor gesicherten, unter Berücksichtigung der Nachbarschaft interpolierten Werten kombiniert werden.

Die Propagierung an die niedrigeren Detailstufen gestaltet sich komplexer und muss im Zuge einer *collapse*-Operation col_v vollzogen werden. Haben sich durch Deformation die globalen Attribute von den in col_v involvierten Vertizes v_t und v_u verändert, müssen die in der Datenstruktur hinterlegten *Offsets* für die korrespondierende *split*-Operation spl_v korrigiert und die Attribute des Zielvertex v über die Minimierung der für col_v anstehenden Kosten neu ermittelt werden.

Um unterscheiden zu können, welche Vertizes modifiziert wurden, wird im Algorithmus ein entsprechend deklariertes Modifizierungsflag auf 1 gesetzt. Dies bedeutet, dass die globalen Vertexattribute und damit das zugehörige lokale Koordinatensystem geändert wurde. Unter Berücksichtigung der editierten Vertizes muss das Referenzkoordinatensystem nun erneut interpoliert werden. Für die Rekonstruktion feinerer Strukturen müssen zusätzlich die mithilfe des alten Systems kodierten *Offsets* übernommen werden. Da für jede *collapse*-Operation eine fixe Nachbarschaft garantiert wird und die Vertexquadriken nach der Simplifizierung aus der Datenstruktur entfernt wurden, müssen beim Kollabieren eines Vertexpaares die jeweiligen Quadriken mittels eines temporären Feldes bestimmt und anschließend deren Summe minimiert werden. Die Berechnung des zugehörigen Simplifizierungsfehlers erfolgt dabei nach dem in Abschnitt 3.5.2.2 geschilderten Prinzip.

Für die an einen modifizierten Vertex angrenzenden Vertizes müssen die Daten ebenfalls angepasst werden. Diese wurden zwar selbst nicht manipuliert, aber aufgrund der Tatsache, dass die *Offsets* der Attribute über ihre direkte Nachbarschaft kodiert wurden, müssen an dieser Stelle die globalen Zusammenhänge wiederhergestellt werden. Hierzu wird eine Überprüfung durchgeführt, bei welcher alle Dreiecke parallel durchlaufen werden. Wird innerhalb eines Dreiecks ein Vertex gefunden, dessen Modifizierungsflag auf den Wert 1 gesetzt wurde, erhalten die Flags der beiden anderen Vertizes den Wert 2. Ein Modifizierungswert von 2 bedeutet hier, dass das lokale Koordinatensystem (also die Nachbarschaft)

bereits angepasst wurde, die globalen Vertexattribute jedoch noch aktualisiert werden müssen. Da die Modifikationen nicht automatisch bis zum *Base Mesh* weitergereicht werden, muss das *progressive Mesh* vor dem Speichern bis auf die gröbste Modellstufe reduziert werden, sodass die Deformationen auch nach erneutem Laden korrekt wiedergeben werden.

3.5.4 Der Adaptionsvorgang

Der Adaptionsalgorithmus, welcher zu gleichen Anteilen von Dipl. Inf. Evgenij Derzapf mit entwickelt wurde, basiert auf dem Verfahren von Derzapf und Guthe [DG12] und kann für die Parallelisierung in mehrere Phasen unterteilt werden. Zunächst muss der Operationsstatus eines Vertex geklärt werden. Soll dieser aufgespalten werden, müssen Referenzen auf die zu generierenden Vertizes vorhanden sein und der zugehörige Fehler über den zu einer Detailstufe festgelegten Schwellwert liegen. Während dieser Prüfung wird zusätzlich die Anzahl der zu generierenden Dreiecke bestimmt. Eine *collapse*-Operation col_v darf nur vollzogen werden, wenn dies aufgrund der Nachbarschaftsverhältnisse zulässig ist. Da die Vertizes über ihre *final vertex IDs* (FVIDs) sortiert wurden, kann ein Vertexpaar v_t und v_u (mit $v_t < v_u$ bzgl. FVID) nur zum Vertex v kollabieren, wenn der Vorgänger von v_u als Zielvertex sich selbst referenziert (im Zuge von col_v wird v_t als Datenkomponente v wiederverwendet) und dessen Nachfolger nicht auf v_u als Zielvertex verweist (vgl. Abbildung 3.10). Ist eine dieser Bedingungen nicht erfüllt, müssen zunächst andere *collapse*-Operationen durchgeführt werden.

Ist das Statusupdate erfolgt, müssen Operationen, die fälschlicherweise als durchführbar gekennzeichnet wurden, gefiltert und entsprechende Flags zurückgesetzt werden. Diese Überprüfung muss erfolgen, um die korrekten Nachbarschaftsverhältnisse garantieren zu können. Dazu werden parallel alle aktiven Dreiecke durchlaufen. In einem Dreieck f darf lediglich der Vertex gesplittet werden, durch dessen inverse *collapse*-Operation während des Simplifizierens der höchste Simplifizierungsfehler erzielt wurde. Für einen *Vertex Split* spl_v muss demnach das lokale Maximum gefunden werden. Für die *collapse*-Operationen gilt, dass nur diejenige mit dem lokalen Kostenminimum durchgeführt werden kann. Zusätzlich darf kein in f befindlicher Vertex kollabieren, wenn ein anderer gesplittet werden soll. Für eindeutige Nachbarschaftsbeziehungen dürfen sich *collapse*- und *split*-Operationen nicht

```
foreach face f in parallel do
    if any_vertex_marked(f, split)
        simplification_error_max = get_max_split_error(f)
        unmark_dependent_splits(f, error_max)
    if any_vertex_marked(f, collapse)
        simplification_error_min = get_min_collapse_error(f)
        unmark_illegal_collapses(f, error_min)
```

Algorithmus 11: Das parallele Eliminieren illegaler Operationen.

gegenseitig behindern. Wird eine Adaption lediglich zwischen den verschiedenen Detailstufen eines Modells vollzogen, sollten derartige Konflikte nicht auftreten. Algorithmus 11 soll diese Vorgehensweise verdeutlichen.

02.
Adaption
Paralleles Kollabieren von Kanten

Im Anschluss werden alle gefilterten *collapse*-Operationen parallel ausgeführt. Bleibt ein Vertex nach dem Editieren unverändert, wird ein Vertexpaar v_t und v_u lediglich auf die alte Position von v verschoben. Wurden jedoch in der direkten Nachbarschaft Modifikationen vorgenommen oder ist einer der Vertizes v_t bzw. v_u editiert worden, müssen Anpassungen an den entsprechenden Vertexattributen realisiert werden. Dies erfordert die Interpolation des lokalen Koordinatensystems, für welches die Anzahl der angrenzenden Vertizes bestimmt und deren Attribute akkumuliert werden (siehe Abschnitt 3.5.3). Wurde das Modifizierungsflag eines Kantenvertex gesetzt, muss dieses an den Zielvertex weitergeleitet werden, um für die gröberen Auflösungen entsprechende Aktualisierungen umsetzen zu können. Das Entfernen von irrelevanten Vertizes und degenerierten Dreiecken erfolgt in einem späteren Schritt. Mit Algorithmus 12 soll die geschilderte parallele Verarbeitung konkretisiert werden.

```
foreach face f in parallel do
    v1, v2, v3 = get_vertices(f)
    atomic_add(acc_adjacent_sum(v1), v2 + v3)
    atomic_add(acc_adjacent_sum(v2), v1 + v3)
    atomic_add(acc_adjacent_sum(v3), v1 + v2)
    atomic_add(adjacent_number(v1), 2)
    atomic_add(adjacent_number(v2), 2)
    atomic_add(adjacent_number(v3), 2)
    q = face_quadric(f)
    atomic_add(vertex_quadric(v1), q)
    atomic_add(vertex_quadric(v2), q)
    atomic_add(vertex_quadric(v3), q)
foreach vertex vu in parallel do
    v = get_target(vu)
    if marked(v, modified) || marked(vu, modified)
        LCS_VT = optimize_quadric(v)
        LCS_VU = optimize_quadric(vu)
        update_split(v, LCS_VT)
        update_split(vu, LCS_VU)
    else
        restore_attributes(v)
    collapse_vertices(v, vu)
```

Algorithmus 12: Die parallele Ausführung der *collapse*-Operationen.

Bevor nun alle *split*-Operationen ausgeführt werden, muss für die neuen Elemente in der Datenstruktur Speicher alloziert werden. Dabei spielt die Reihenfolge in welcher die Dreiecke eingefügt werden keine Rolle, die Vertizes müssen hingegen im Bezug auf ihre FVID einsortiert werden. Verfügen die Buffer (*Index*- und FVID-Buffer) über ausreichende Kapazitäten, können die Dreiecke direkt eingefügt werden. Für eine genaue Abschätzung des Platzbedarfs muss die komplette Anzahl aller zu generierenden Dreiecke ermittelt werden. Sowohl das Bestimmen der Dreiecksanzahl, als auch die Sortierung der Vertizes nach den FVIDs, kann mithilfe einer Präfixsummen-Berechnung verwirklicht werden [SHZO07]. Für die Sortierung wird zusätzlich ein neuer Buffer angelegt, in welchem die vorhanden Daten an der korrekten Position abgelegt werden. Ergänzend sei erwähnt, dass während dieses Prozesses die ungültig gewordenen *collapse*-Vertizes entfernt werden. Um das Allozieren oder Freigeben von Speicher pro Iteration zu vermeiden, wird immer etwas mehr Speicher alloziert bzw. eine Reserve beibehalten. Algorithmus 13 soll diesen Schritt veranschaulichen.

```
face_sum = prefix_sum(face_cnt)
if need_face_buffer_resize()
      resize_face_buffer()
vertex_sum = prefix_sum(vertex_cnt)
if need_vertex_buffer_resize()
      resize_vertex_buffer()
foreach vertex v in parallel do
      new_pos = vertex_sum[v]
      next_pos = vertex_sum[v+1]
      if new_pos != next_pos
            copy_vertex(v, new_pos)
```

Algorithmus 13: Vorgehensweise zum Speichermanagement.

Mit der Durchführung der *split*-Operationen werden die in der Datenstruktur hinterlegten *Offsets* dazu verwendet, die globalen Attribute der Vertizes v_t und v_u zu rekonstruieren. Zu diesem Zweck wird nochmals das lokale Koordinatensystem mithilfe der Nachbarschaft bestimmt (siehe Abschnitt 3.5.3.1). Zunächst wird jedoch die anfänglich beim Statusupdate ermittelte Anzahl der zu generierenden Dreiecke abgerufen, um selbige zu erzeugen und in den Buffer aufzunehmen. Erst wenn diese in die Datenstruktur aufgenommen wurden, werden die relevanten Vertexattribute berechnet. Um die *Thread*-Leistung optimal auszunutzen, werden die *split*-Operationen kompaktiert, sodass pro *Thread* ein *Vertex Split* ausgeführt werden kann [SHZO07]. Algorithmus 14 zeigt die entsprechenden parallelen Einzelschritte.

Nachdem alle Operationen ausgeführt wurden, muss abschließend die Konnektivität zwischen den einzelnen Komponenten hergestellt werden. Dies bedeutet, dass die Vertexindizes von Dreiecken aktualisiert werden müssen, die sich in der direkten Nachbarschaft einer *collapse*- bzw. *split*-Operation befanden, wo im Zuge dessen Vertizes entfernt oder

```
compact(splits)
foreach split vertex v in parallel do
    v_u = v + 1
    split_vertex(v, v_u)
    append_faces(v, face_sum[v])
foreach face f in parallel do
    v_1, v_2, v_3 = get_vertices(f)
    atomic_add(acc_adjacent_sum(v_1), v_2 + v_3)
    atomic_add(acc_adjacent_sum(v_2), v_1 + v_3)
    atomic_add(acc_adjacent_sum(v_3), v_1 + v_2)
    atomic_add(adjacent_number(v_1), 2)
    atomic_add(adjacent_number(v_2), 2)
    atomic_add(adjacent_number(v_3), 2)
foreach split vertex v in parallel do
    LCS_VT = acc_adjacent_sum(v) / adjacent_number(v_t)
    LCS_VU = acc_adjacent_sum(v_u) / adjacent_number(v_u)
    calc_attributes(v, v_u, LCS_VT, LCS_VU)
```

Algorithmus 14: Die parallele Ausführung der *split*-Operationen.

hinzugefügt wurden. Für das Rendering müssen zusätzlich noch alle degenerierten Dreiecke entfernt werden, was mit einer *In-Place-Compaction* (siehe Abschnitt 3.4.2.2) realisiert werden kann.

Die für das Editieren und für die parallele Adaption verwendeten Elemente der dynamische Datenstruktur sind in Tabelle 3.7 aufgelistet. Im *Vertex Buffer* sind die Vertexattribute enthalten und die Konnektivität wird durch den *Index Buffer* repräsentiert. Beide Buffer sind für das Rendering erforderlich.

buffers	**elements**	**bytes per entry**
faces	index VBO	12
	FVIDs	12
vertices	vertex VBO (×2)	$8k$
	vertex ID (×2)	8
	modified flag (×2)	2
	collapse target (×2)	8
	next split & collapse (×2)	16
temporary	vertex count	4
	face count	4
	vertex prefix sum	4
	face prefix sum	4
	vertex quadric	$2k^2 + 6k + 4$

Tabelle 3.7: Elemente der dynamischen Datenstruktur, wobei k für die Anzahl der berücksichtigten Vertexattribute steht.

3.5.5 Ergebnisse

Zur näheren Verfahrensanalyse stand ein Testsystem, bestehend aus einem 3.333 GHz Intel Core i7-980X Prozessor, einem 6 GB DDR3-1333 Hauptspeicher und einer NVIDIA GeForce GTX 580 Grafikkarte (841/4204 MHz), zur Verfügung. Für die Implementierung des parallelen Simplifizierungs- und Adaptionsalgorithmus wurde CUDA und für das Rendering OpenGL verwendet. Tabelle 3.8 und 3.9 enthalten die Informationen zu den betrachteten Modellen, den generierten *progressiven Meshes* und zum Laufzeitverhalten des Verfahrens während der Modellierungsphase.

Modell	v_{max}	f_{max}	IFS
Horse	48,485	96,966	2.7 MB
Armadillo	172,974	345,944	7.9 MB
St. Dragon	437,645	871,414	19.9 MB
Welsh Dragon	1,105,352	2,210,673	50.5 MB
Dragon	3,609,455	7,218,906	165.2 MB

Tabelle 3.8: Die zur Evaluation verwendeten Modelle inklusive der maximalen Anzahl von enthaltenen Vertizes und Dreiecken sowie der benötigte Speicherbedarf.

	Simplifizierung				Modellierung/Adaption		
Modell	PM	memory	time (s)	kOp/s	split	collapse	update
Horse	5.4 MB	41.3 MB	0.3	162	595	443	384
Armadillo	19.1 MB	147.3 MB	0.6	288	1181	810	631
St. Dragon	48.3 MB	372.2 MB	1.4	313	1201	882	674
Welsh Dragon	122.3 MB	941.6 MB	3.2	346	986	381	336
Dragon	399.0 MB	1872.6 MB	10.4	347	760	379	335

Tabelle 3.9: Performanz während der Simplifizierung, Modellierung und Adaption. Die Angaben der letzten drei Spalten beziehen sich auf die Anzahl von durchführbaren Operationen pro Sekunde (kOp/s).

Für alle angegebenen Modelle wurde eine Attributanzahl (Vertexposition und Normale) von $k = 6$ berücksichtigt. Die Originalmodelle enthalten dabei eine maximale Anzahl von v_{max} Vertizes und f_{max} Dreiecken. Da die *progressiven Meshes* nicht komprimiert werden, wird für diese im Gegensatz zum Original (IFS) ungefähr der doppelte Speicher benötigt. Der maximale Speicherbedarf ist im Vergleich zum geladenenen IFS bei den größeren Modellen um den Faktor 11 und bei den kleineren Objekten um den Faktor 16 höher. Der angegebene Speicher ist lediglich zu Beginn für die Simplifizierung erforderlich und kann während der Modellierung um ca. 1/3 reduziert werden, da die Kantenstruktur entfällt. Wird ein *progressives Mesh* auf die höchste Detailstufe verfeinert, werden Werte ähnlich des maximalen Speicherbedarfs erreicht.

Im Vergleich zum *Basis-Simplifizierer* [GDG11] sinkt die Performanz des für die Modellierung modifizierten Simplifizierers um den Faktor 5,8. Hierfür gibt es zwei wesentliche

Gründe: Zum einen können aufgrund der zu gewährleistenden Nachbarschaftsverhältnisse knapp die Hälfte der *collapse*-Operation parallel ausgeführt und zum anderen muss die Datenstruktur für das *progressive Mesh* erstellt werden. Die Leistung des Adaptionalgorithmus ist gegenüber dem Verfahren von Derzapf und Guthe [DG12] um den Faktor 9,6 langsamer. Dies ist wiederum teilweise der zu garantierenden Nachbarschaft geschuldet, welche die Anzahl der parallel ausführbaren Operationen auf ca. 1/6 beschränkt. Andererseits wird für die unkomprimierten Daten mehr Speicher benötigt und die Transformationen zwischen lokalen bzw. globalen Attributen bremsen die Adaption zusätzlich aus. Des Weiteren enthalten größere Modelle, die nicht vorverarbeitet wurden, oftmals koplanare Dreiecke. Diese Tatsache führt dazu, dass im entwickelten Ansatz viele *collapse*-Operationen blockiert werden, da lediglich die Kanten kollabieren dürfen, welche innerhalb ihrer Nachbarschaft über das lokale Fehlerminimum verfügen. Mit einer entsprechenden Problemlösung könnte die Performanz letztlich erhöht werden.

Die Modellierungsperformanz ist dem Verfahren von Marinov [MBK07] ebenbürtig. Allerdings können mit dem hier beschriebenen Verfahren Modelle auf einer beliebigen Detailstufe deformiert und die Modifikation direkt zu allen weiteren LODs übertragen werden. Frühere Techniken, wie beispielsweise die Methode von Zorin et al. [ZSS97] sind lediglich auf Modelle mit einer *subdivision connectivity* anwendbar. Bei der Editierung eines *Meshes*, welches über *Laplace-Operatoren* kodiert wurde [SCOL*04], wird eine geglättete Objektoberfläche für die Modellierung genutzt, wobei in einem abschließenden Schritt die vollzogenen Veränderungen auf das eigentliche Modell transferiert werden. Der Nachteil dieser Vorgehensweise liegt in der Beschränkung der manipulierbaren Objektregionen (ROIs) mit maximal 100K Vertizes, welche lediglich zu interaktiven Frameraten modifiziert werden können. Hingegen können mit der entwickelten Methode Modelle, die mehrere Millionen Dreiecke umfassen, unabhängig von der Größe der ROIs, zu Echtzeit-Frameraten editiert werden. Die Abbildungen 3.9 und 3.15 zeigen einige der entsprechend editierten *progressiven Meshes*.

3.5.6 Zusammenfassung

In den letzten Kapiteln wurde ein Algorithmus diskutiert, mit welchem *progressive Meshes* parallel generiert sowie editiert werden können. Als Eingabe wird ein *Indexed Face Set* (IFS) gelesen, welches im Zuge einer Simplifizierungsphase in ein *progressives Mesh* bzw. in eine adäquate Datenstruktur umgewandelt wird. Diese kann anschließend dazu verwendet werden, ein Modell in einer beliebigen Auflösung zu deformieren. Die Modifikationen werden automatisch an die feineren Detailstufen übertragen, indem mithilfe eines interpolierten Referenzkoordinatensystems lokale *Offsets* berechnet und in Verbindung mit den ausführbaren *Vertex Split* Operationen kodiert werden. Darüber hinaus werden die gröberen Modellstrukturen bis hin zum *Base Mesh* während der Laufzeit aktualisiert, ohne dabei auf extra im Speicher hinterlegte Daten zurückzugreifen. Durch eine derartige

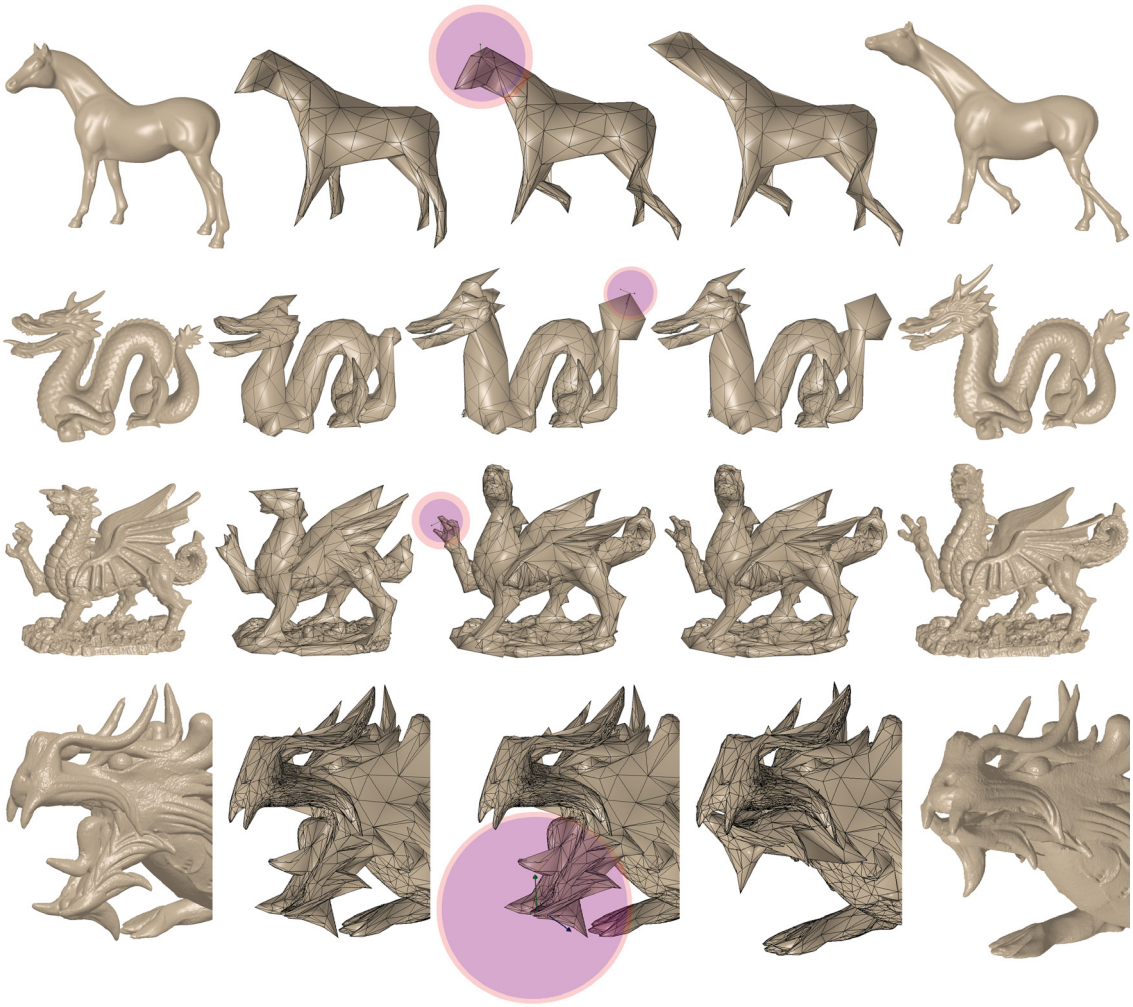


Abbildung 3.15: Die für die Evaluation verwendeten *progressiven Meshes*. Von links nach rechts: Originalmodell, drei Aufnahmen vor, während und nach dem Editiermodus, wobei die violetten Kugeln die ROIs veranschaulichen und das entgültige *progressive Mesh* in der höchsten Auflösungsstufe.

speicherschonende Simplifizierung (engl. *memoryless simplification*) kann über die gesamte Editierungsphase hinweg ein gültiges *progressives Mesh* garantiert werden.

Eine Beschränkung des Verfahrens besteht darin, dass die geladenen Modelle aktuell nur einige Millionen Dreiecke enthalten dürfen. Zukünftig sollte dieser Algorithmus verbessert werden, um die Verarbeitung größerer Modelle zu ermöglichen. Dies könnte beispielsweise über *Out-of-Core* oder Komprimierungstechniken realisiert werden. Als nachteilig könnte ebenfalls die strikte lokale Ordnung der durch die anfängliche Simplifizierung ermittelten Operationen angesehen werden. Dies ist zwar für die Umsetzung von Animationen notwendig, bei größeren Deformationen könnte jedoch eine partielle Re-Simplifizierung erstrebenswert sein.

Zudem ist dieses Verfahren, wie alle Techniken zur Multiskalen-Modellierung, auf die Modifikation der Geometrie beschränkt. Änderungen an der Konnektivität bzw. topologische Veränderungen, bei welchen Geometrie hinzugefügt oder entfernt werden kann, machen nach derzeitigem Stand eine komplette Reproduktion des *progressiven Meshes* sowie der dynamischen Datenstruktur erforderlich. Zwar können nach einer Deformation die damit in Verbindung stehenden Operationen lokalisiert und entsprechend angepasst werden, der Aufwand um Modifikationen an der Konnektivität mit einzubeziehen wäre jedoch unweit höher als die reine geometrische Anpassung.

KAPITEL 4

Zusammenfassung und Fazit

In vielen Anwendungsgebieten der Computergrafik spielt die Verarbeitung von hochwertigen 3D-Modellen eine große Rolle. Besonders im Bereich der Spieleentwicklung und bei Animationsfilmen werden die 3D-Modelle dazu verwendet komplette Welten zu kreieren. Der Handlung entsprechend sollen die virtuellen Welten eine bestimmte Grundstimmung transportieren, sodass sich Spieler oder Zuschauer leichter in die dargebotene Geschichte einfinden oder sich mit den Hauptfiguren identifizieren kann. Um dies zu ermöglichen wird ein hohes Maß an Realismus vorausgesetzt, was dazu führt, dass die 3D-Modelle immer detaillierter werden und damit das zu verarbeitende Datenvolumen enorm zunimmt. Im Zuge dieses Realitätsanspruches dienen zudem vermehrt real vorhandene Orte, Gebäude oder Personen als Vorlage, welche digitalisiert und im Anschluss nachbearbeitet oder modifiziert werden müssen. Für die Digitalisierung kann auf Laserscanner zurückgegriffen werden. Da diese jedoch teuer und unhandlich sind, bedarf es kostengünstiger Alternativen. Sind entsprechende 3D-Modelle verfügbar, werden Programme benötigt, mit deren Hilfe die korrespondierenden Datenmengen reduziert und editiert werden können.

Aufgrund dieser Tatsache wurde im Rahmen dieser Dissertation unter anderem eine Methode zur 3D-Rekonstruktion von Objekten und Szenen aus zweidimensionalen Kamerabildern entwickelt (siehe Kapitel 2.4) sowie ein Verfahren zur Simplifizierung (siehe Kapitel 3.4) und Modellierung (siehe Kapitel 3.5) von dreidimensionalen Modellen implementiert. Da die Rekonstruktion auf die Verarbeitung reziproker Bildpaare zurückgeführt werden kann, wurde zusätzlich eine Technik für die Kompression von Bilddaten realisiert (siehe Kapitel 2.7), welche basierend auf einer Wavelet-Transformation und einer abgewandelten Form des *Zerotree Coding* hohe Kompressionsraten ohne eine sichtbare Reduktion der visuellen Qualität erzielt. Während der Implementierung der verschiedenen Verfahren wurde großer Wert auf die Parallelisierung der einzelnen Berechnungsschritte gelegt. In diesem Sinne musste entsprechend dem Anwendungsfall eine geeignete sowie effiziente Datenstruktur konzipiert werden, auf deren Grundlage die Daten parallel auf der Grafikkarte verarbeitet werden können. In Hinblick auf die Simplifizierung und der Möglichkeit des Editierens dreidimensionaler Modelle wurden die jeweiligen Datenstrukturen so optimiert,

dass die Datenverarbeitung in Echtzeit ausgeführt und somit die Animation eines 3D-Modells sowie die Simulation von komplexen Bewegungsabläufen verwirklicht werden kann.

Für die 3D-Rekonstruktion wurden folgende Ziele definiert (siehe Kapitel 2.3):

- die Entwicklung eines hochqualitativen, parallelen Verfahrens zur 3D-Rekonstruktion
- die Generierung einer effizienten und speicherschonenden Datenstruktur zur Verwaltung relevanter Daten
- die Verwendung eines adäquaten Fehlermaßes zur Minimierung von Berechnungsfehlern
- die Verarbeitung von hochauflösenden Bilddaten
- die Rekonstruktion feinerer und spiegelnder Objektstrukturen

Durch die Verschmelzung der Korrespondenzanalyse mit der Suche nach dem *Best Match*, welche durch ein Verfahren nach dem Prinzip der *Belief Propagation* erreicht wurde, konnte zum einen die Datenstruktur bzw. der für den Algorithmus benötigte Speicher reduziert und zum anderen die Effizienz bzw. Performanz der für die Rekonstruktion notwendigen Tiefenabschätzung gesteigert werden. Zudem wurde gezeigt, dass die Tiefenanalyse umso präziser wird, je mehr reziproke Bildpaare für die Generierung einer Tiefenkarte verwendet werden. In Verbindung mit dem für das entwickelte Verfahren definierte Fehlermaß, wurde die Anzahl der Fehlinterpretationen reduziert und somit der Bildung von Artefakten nachhaltig vorgebeugt. Eine Beschränkung der Rekonstruktionsmethode besetht darin, dass mögliche Lücken im rekonstruierten Polygonnetz einfach geschlossen und nicht näher untersucht werden. An dieser Stelle könnte das Verfahren verbessert werden, indem Löcher im Modell erkannt und diese entsprechend der in der näheren Umgebung vorherrschenden Strukturen remodelliert werden.

Für die Simplifizierung und Modellierung dreidimensionaler Modelle wurden folgende Zielesetzungen deklariert (siehe Kapitel 3.3):

- der Entwurf eines hochqualitativen, parallelen Simplifizierungsalgorithmus, basierend auf dem Prinzip der *Vertex Pair Contraction*
- die effiziente Berechnung der optimalen Vertexposition, in welche eine durch ein Vertexpaar definierte Kante kollabiert
- die Verwendung eines adäquaten Fehlermaßes zur Minimierung des Simplifizierungsfehlers
- die Simplifizierung von hochgradig detaillierten Modellen in weniger als einer Sekunde

- das interaktive Editieren von großen Datenmodellen (10 bis 15 Frames pro Sekunde)
- die in Echtzeit vollzogene Propagation der Modifikationen über alle Detailstufen hinweg (mindestens 25 Frames pro Sekunde)
- die Generierung einer hinreichenden, für die Modellierung notwendigen Datenstruktur während der parallelen *Mesh*-Simplifizierung
- die Entwicklung einer Datenstruktur, mit welcher Modelle auf verschiedenen Detailstufen bearbeitet und zur Erstellung von Animationen verwendet werden können

Für die Generierung von statischen *Level-of-Detail* (LOD) wurde ein parallel auf der Grafikkarte ausführbarer Algorithmus entworfen und im Anschluss für die Multiskalen Modellierung optimiert. Der entwickelte Simplifizierungsalgorithmus entspricht der ersten parallelen Umsetzung auf dem Gebiet der *Vertex Pair Contraction* und ist in der Lage, mehrere zu einem 3D-Modell gehörende Detailsstufen in weniger als einer Sekunde zu erzeugen. Um den geometrischen Fehler beim Kollabieren von Kanten zu minimieren, wurden die metrischen Fehlerquadriken für die Berechnungen der optimalen Vertexposition eingesetzt, zu welcher die beiden Kantenvertizes zusammengefasst werden. Während der Positionsbestimmung können in deren Abhängigkeit die entsprechenden Kosten für die anstehende *collapse*-Operation ermittelt und analysiert werden. Für die Umsetzung der Modellierungsmöglichkeit musste die Datenstruktur des Simplifizierer in geeigneter Form angepasst werden. Das erweiterte Verfahren wird dazu verwendet, um die feste Reihenfolge der durchzuführenden *collapse*- und *split*-Operationen festzulegen sowie die zu deren korrekter Ausführung benötigten Daten zu sammeln und zu sichern. Um Modifikationen an einem 3D-Modell durchführen zu können, kann das Programm in den Editiermodus versetzt werden, in welchem der Anwender ein beliebiges Objekt auf verschiedenen Detailstufen interaktiv manipulieren kann. Die vollzogenen Veränderungen werden anschließend automatisch in Echtzeit zu allen anderen Detailstufen propagiert und die in der Datenstruktur hinterlegten Informationen entsprechend aktualisiert. Da derzeit lediglich Modelle mit einigen Millionen Dreiecken verarbeitet werden können, sollte der Algorithmus durch Komprimierungstechniken optimiert oder als *Out-of-Core*-Anwendung umgesetzt werden. Zudem beschränken sich die durchführbaren Modifikationsmöglichkeiten auf die Rotation und Translation bestehender Geometrie. Topologische Veränderungen, wie sie beispielsweise durch das Hinzufügen oder Entfernen von Geometriedaten entstehen, wurden nicht berücksichtigt. Dies hätte hinreichende Anpassungen an der Datenstruktur sowie eine Re-Simplifizierungsphase des Modells zur Folge, um die Konnektivität zwischen den im Dreiecksnetz enthaltenen Komponenten gewährleisten zu können.

Zusammenfassend kann festgehalten werden, dass alle definierten Ziele erfolgreich umgesetzt und erreicht wurden. Durch die verschiedenen Tests konnten akzeptable bis sehr gute Ergebnisse produziert und potenzielle Optimierungsmöglichkeiten aufgezeigt werden, die in naher Zukunft realisiert werden sollen.

Abkürzungsverzeichnis

BRDF	**B**idirectional **R**eflectance **D**istribution **F**unction (deut. *Bidirektionale Reflektanzverteilungsfunktion*)
BSP	**B**inary **S**pace **P**artitioning (deut. *Binäre Raumpartitionierung*)
CC	**C**ross **C**orrelation (deut. *Kreuzkorrelation*)
CG	**C**onjugate **G**radients
CPU	**C**entral **P**rocessing **U**nit (deut. *Zentrale Verarbeitungseinheit/Prozessor*)
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
CWT	**C**ontinuous/**C**omplex **W**avelet **T**ransformation (deut. *Kontinuierliche/Komplexe Wavelet-Transformation*)
DDR SDRAM	**D**ouble **D**ata **R**ate **S**ynchronous **D**ynamic **R**andom-**A**ccess **M**emory
DFT	**D**iscrete **F**ourier **T**ransformation (deut. *Diskrete Fourier-Transformation*)
DWT	**D**iscrete **W**avelet **T**ransformation (deut. *Diskrete Wavelet-Transformation*)
EZW	**E**mbedded **Z**erotree **W**avelet
FFT	**F**ast **F**ourier **T**ransformation (deut. *Schnelle Fourier-Transformation*)
FT	**F**ourier **T**ransformation (deut. *Fourier-Transformation*)
FVID	**F**inal **V**ertex **ID**
GPU	**G**raphics **P**rocessing **U**nit (deut. *Grafische Verarbeitungseinheit/Grafikprozessor*)
HDD	**H**ard **D**isk **D**rive (deut. *Festplattenlaufwerk*)

HDR	**H**igh **D**ynamic **R**ange
IFS	**I**ndexed **F**ace **S**et
IRLS	**I**teratively **R**eweighted **L**east **S**quares
JPEG	**J**oint **P**hotographic **E**xperts **G**roup
LOD	**L**evel **O**f **D**etail (deut. *Detailstufen*)
MRA	**M**ulti **R**esolution **A**nalysis (deut. *Multiskalen Analyse*)
MRD	**M**ulti **R**esolution **D**eformation (deut. *Multiskalen Deformation*)
MRM	**M**ulti **R**esolution **M**odeling (deut. *Multiskalen Modellierung*)
NCC	**N**ormalized **C**ross **C**orrelation (deut. *Normalisierte Kreuzkorrelation*)
PSNR	**P**eak **S**ignal-to-**N**oise **R**atio (deut. *Spitzen-Signal-Rausch-Verhältnis*)
ROI	**R**egion **O**f **I**nfluence (deut. *Einflussbereich*)
SAD	**S**um of **A**bsolute **D**ifferences (deut. *Summe der absoluten Differenzen*)
SSD	**S**um of **S**quared **D**ifferences (deut. *Summe der quadratischen Differenzen*)
SURF	**S**peeded **U**p **R**obust **F**eatures
VBO	**V**ertex **B**uffer **O**bject
WT	**W**avelet **T**ransformation (deut. *Wavelet-Transformation*)

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[AGD11] Azad P., Gockel T., Dillmann R.: *Computer Vision: das Praxisbuch*. Elektor-Verlag, Aachen, 2011.

[Ana89] Anandan P.: A computational framework and an algorithm for the measurement of visual motion. *International Journal of Computer Vision 2*, 3 (January 1989), 283–310–310.

[BAC96] Beers A. C., Agrawala M., Chaddha N.: Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 373–378.

[Bar04] Bartsch H.-J.: *Taschenbuch mathematischer Formeln*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2004.

[BK01] Boykov Y., Kolmogorov V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26* (2001), 359–374.

[BKBH09] Bolitho M., Kazhdan M., Burns R., Hoppe H.: Parallel poisson surface reconstruction. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I* (Berlin, Heidelberg, 2009), ISVC '09, Springer-Verlag, pp. 678–689.

[Bän02] Bäni W.: *Wavelets. Eine Einführung für Ingenieure*. Oldenbourg Verlag, 2002.

[Büs10] Büsing C.: *Graphen- und Netzwerkoptimierung*. Spektrum Akademischer Verlag, 2010.

[BT08] Bhusnurmath A., Taylor C. J.: Graph cuts via l norm minimization. *IEEE Trans. Pattern Anal. Mach. Intell.* (2008), 1866–1871.

[CSE00] Christopoulos C., Skodras A., Ebrahimi T.: The jpeg2000 still image coding system: An overview. In *IEEE Transactions on Consumer Electronics* (2000), vol. 16, pp. 1103–1127.

[CT65] Cooley J., Tukey J.: An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation 19*, 90 (1965), 297–301.

[Dau92] Daubechies I.: *Ten Lectures on wavelets*. Society for Industrial and Applied Mathematics, 1992.

[DCH05a] Diverdi S., Candussi N., Höllerer T.: *Real-time Rendering with Wavelet-Compressed Multi-Dimensional Datasts on the GPU*. Tech. rep., University of California, Santa Barbara, 2005.

[DCH05b] DiVerdi S. J., Candussi N., Höllerer T.: *Real-time Rendering with Wavelet-Compressed Multi-Dimensional Datasets on the GPU*. Tech. rep., University of California, Santa Barbara, 2005.

[DG12] Derzapf E., Guthe M.: Dependency free parallel progressive meshes. *Computer Graphics Forum 31* (December 2012), 2288–2302.

[DKP05] Dixit N., Keriven R., Paragios N.: Gpu-cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction, 2005.

[DMG10] Derzapf E., Menzel N., Guthe M.: Parallel view-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 53–62.

[DT07] DeCoro C., Tatarchuk N.: Real-time mesh simplification using the gpu. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 161–166.

[FB88] Forsey D. R., Bartels R. H.: Hierarchical b-spline refinement. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (1988), SIGGRAPH '88, ACM, pp. 205–212.

[Fen03] Fenney S.: Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 84–91.

[FF56] Ford L. R., Fulkerson D. R.: Maximal flow through a network. *Canadian Journal of Mathematics 8* (1956), 399–404.

[FI96] Franke U., I. K.: Fast stereo based object detection for stop&go traffic. In *Proceedings of the 1996 IEEE, Intelligent Vehicles Symposium* (September 1996), pp. 339–344.

[FJ00] Franke U., Joos A.: Real-time stereo vision for urban traffic scene understanding. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE* (2000), pp. 273–278.

[FP12] Forsyth D. A., Ponce J.: *Computer Vision: A Modern Approach*. Pearson Education, 2012.

[Gar] Garg S.: The image compression benchmark. `http://http://www.imagecompression.info/test_images/` (30.10.2012).

[Gar99] Garland M.: Multiresolution modeling: Survey & future opportunities. *Proceedings of the Eurographics '99 – State of the Art Reports* (1999), 111–131.

[GDG] Grund N., Derzapf E., Guthe M.: Parallel progressive mesh editing. Unpublished.

[GDG11] Grund N., Derzapf E., Guthe M.: Instant level-of-detail. In *Vision, Modeling, and Visualization (VMV2011)* (2011), pp. 293–299.

[GG] Grund N., Guthe M.: Multi-resolution helmholtz stereopsis. Unpublished.

[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 209–216.

[GH98] Garland M., Heckbert P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98* (1998), pp. 263–269.

[GMG10] Grund N., Menzel N., Guthe M.: High-quality wavelet compressed textures for real-time rendering. In *WSCG 2010 Communication Papers Proceedings* (2010), pp. 207–212.

[GS02] Garland M., Shaffer E.: A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization '02* (2002), pp. 117–124.

[GSS99] Guskov I., Sweldens W., Schröder P.: Multiresolution signal processing for meshes. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), SIGGRAPH '99, pp. 325–334.

[GT88] Goldberg A. V., Tarjan R. E.: A new approach to the maximum-flow problem. In *Journal of the Association for Computing Machinery (ACM)* (October 1988), vol. 35, pp. 921–940.

[Gut84] Guttman A.: R-trees: A dynamic index structure for spatial searching. In *International Conference on Management of Data* (1984), ACM, pp. 47–57.

[HH10] He Z., Hong B.: Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010* (2010), IEEE, pp. 1–10.

[HIG02] Hirschmüller H., Innocent P. R., Garibaldi J.: Real-time correlation-based stereo vision with reduced border errors. *Int. J. Comput. Vision 47* (April 2002), 229–246.

[HZ03] Hartley R., Zisserman A.: *Multiple View Geometry in Computer Vision.* Cambridge University Press, 2003.

[Jäh05] Jähne B.: *Digitale Bildverarbeitung.* Springer Verlag, 2005.

[Kaz05] Kazhdan M.: Reconstruction of solid models from oriented point sets. In *Proceedings of the third Eurographics symposium on Geometry processing* (2005), SGP '05.

[KBH06] Kazhdan M., Bolitho M., Hoppe H.: Poisson surface reconstruction. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing* (2006), Eurographics Association, pp. 61–70.

[KCVS98] Kobbelt L., Campagna S., Vorsatz J., Seidel H.-P.: Interactive multi-resolution modeling on arbitrary meshes. In *SIGGRAPH* (1998), pp. 105–114.

[KKDH07] Kazhdan M., Klein A., Dalal K., Hoppe H.: Unconstrained isosurface extraction on arbitrary octrees. In *Proceedings of the fifth Eurographics symposium on Geometry processing* (Aire-la-Ville, Switzerland, Switzerland, 2007), SGP '07, Eurographics Association, pp. 125–133.

[KLCL05] Kim J. C., Lee K. M., Choi B. T., Lee S. U.: A dense stereo matching using two-pass dynamic programming with generalized ground control points. In *Proceedings IEEE International Conference on Computer Vision and Pattern Recognition, Vol. II* (2005), IEEE Computer Society, pp. 1075–1082.

[KO94] Kanade T., Okutomi M.: A stereo matching algorithm with an adaptive window: Theory and experiment. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence* (September 1994), vol. 16, pp. 920–932.

[KVS99] Kobbelt L., Vorsatz J., Seidel H.-P.: Multiresolution hierarchies on unstructured triangle meshes. *Comput. Geom. 14*, 1-3 (1999), 5–24.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 163–169.

[Lin00] Lindstrom P.: Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), SIGGRAPH '00, pp. 259–262.

[LOH08] Lee S.-B., Oh K.-J., Ho Y.-S.: Segment-based multi-view depth map estimation using belief propagation from dense multi-view video. In *Proceedings of the 3DTV Conference* (May 2008), pp. 193–196.

[LSCO*04] Lipman Y., Sorkine O., Cohen-Or D., Levin D., Rössl C., Seidel H.-P.: Differential coordinates for interactive mesh editing. In *In Proceedings of Shape Modeling International* (2004), Society Press, pp. 181–190.

[LT97] Low K.-L., Tan T.-S.: Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics* (1997), pp. 75–ff.

[Lue01] Luebke D. P.: A developer's survey of polygonal simplification algorithms. *IEEE Comp. Graph. Appl. 21* (2001), 24–35.

[Mal98] Mallat S.: *A Wavelet Tour of Signal Processing.* Academic Press, 1998.

[MBK07] Marinov M., Botsch M., Kobbelt L.: Gpu-based multiresolution deformation using approximate normal field reconstruction. *journal of graphics, gpu, and game tools 12*, 1 (2007), 27–46.

[MKZB01] Magda S., Kriegman D. J., Zickler T., Belhumeur P. N.: Beyond lambert: Reconstructing surfaces with arbitrary brdfs. In *Proceedings of the 8th IEEE International Conference on Computer Vision* (June 2001), vol. 2, pp. 391–399.

[MLM*07] Matthies, Larry, Maimone, Mark, Johnson, Andrew, Cheng, Yang, Willson, Reg, Villalpando, Carlos, Goldberg, Steve, Huertas, Andres, Stein, Andrew, Angelova, Anelia: Computer vision on mars. *International Journal of Computer Vision 75*, 1 (October 2007), 67–92.

[OGH*98] Ohm J.-R., Grüneberg K., Hendriks E., M. E. I., Kalivas D., Karl M., Papadimatos D., Redert A.: A realtime hardware system for stereoscopic videoconferencing with viewpoint adaptation. In *Signal Processing: Image Communication* (1998), pp. 147–171.

[PB99] Petrou M., Bosdogianni P.: *Image Processing: The Fundamentals*. John Wiley & Sons Inc., 1999.

[PH97] Popović J., Hoppe H.: Progressive simplicial complexes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 217–224.

[Pra01] Pratt W. K.: *Digital Image Processing*. John Wiley & Sons Inc., 2001.

[PTVF07] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.: *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.

[RB93] Rossignac J., Borel P.: Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications* (Berlin, June 1993), Falcidieno B., Kunii T., (Eds.), Springer Verlag, pp. 455–465.

[S3T98] S3tc directx 6.0 standard texture compression. S3 Inc., 1998.

[SB04] Samavati F. F., Bartels R. H.: Local filters of b-spline wavelets. In *Proceedings of International Workshop on Biometric Technologies 2004* (2004), pp. 105–110.

[SCD*06] Seitz S. M., Curless B., Diebel J., Scharstein D., Szeliski R.: A comparison and evaluation of multi-view stereo reconstruction algorithms, 2006.

[Sch05] Schreer O.: *Stereoanalyse und Bildsynthese*. Springer-Verlag, 2005.

[SCOL*04] Sorkine O., Cohen-Or D., Lipman Y., Alexa M., Rössl C., Seidel H.-P.: Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2004), SGP '04, pp. 175–184.

[SFYC96] Shekhar R., Fayyad E., Yagel R., Cornhill J. F.: Octree-based decimation of marching cubes surfaces. In *Proceedings of the 7th conference on Visualization '96* (Los Alamitos, CA, USA, 1996), VIS '96, IEEE Computer Society Press, pp. 335–ff.

[SG01] Shaffer E., Garland M.: Efficient adaptive simplification of massive meshes. In *Proceedings of the conference on Visualization '01* (2001), pp. 127–134.

[SG07] Sinop A. K., Grady L.: A seeded image segmentation framework unifying graph cuts and random walker which yields a new algorithm. In *Computer*

Vision, 2007. ICCV 2007. IEEE 11th International Conference on (2007), IEEE, pp. 1–8.

[Sha01] Shapiro J. M.: Readings in multimedia computing and networking. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, ch. Embedded Image Coding Using Zerotrees of Wavelet Coefficients, pp. 124–141.

[SHM06] Stein A. N., Huertas A., Matthies L.: Attenuating stereo pixel-locking via affine window adaptation. In *In IEEE International Conference on Robotics and Automation* (2006), pp. 914–921.

[SHZO07] Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan primitives for gpu computing. In *GRAPHICS HARDWARE 2007* (2007), Association for Computing Machinery, pp. 97–106.

[SS02] Scharstein D., Szeliski R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision 47* (2002), 7–42.

[Ste04] Stein F.: Efficient computation of optical flow using the census transform. In *Proceedings of the 26th DAGM Symposium* (2004), vol. 3175 of *Lecture Notes in Computer Science*, Springer, pp. 79–86.

[SW03a] Schaefer S., Warren J.: Adaptive vertex clustering using octrees. In *SIAM Geometric Design and Computing* (2003).

[SW03b] Schneider J., Westermann R.: Compression domain volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), VIS '03, IEEE Computer Society, pp. 293–300.

[SW03c] Seber G. A. F., Wild C. J.: *Nonlinear Regression*. Wiley Interscience, 2003.

[Tau95] Taubin G.: A signal processing approach to fair surface design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), SIGGRAPH '95, ACM, pp. 351–358.

[TM01] Taubman D. S., Marcellin M. W.: Jpeg2000: Image compression fundamentals, standards and practice. In *Kluwer International Series in Engineering and Computer Science* (2001), Kluwer Academic Publishers.

[TMJ98] Tanner C. C., Migdal C. J., Jones M. T.: The clipmap: A virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 151–158.

[VC95] Venkateswar V., Chellappa R.: Hierarchical stereo and motion correspondence using feature groupings. *Int. J. Comput. Vision 15*, 3 (July 1995), 245–269.

[VN08] Vineet V., Narayanan P. J.: Cuda cuts: Fast graph cuts on the gpu. *Computer Vision and Pattern Recognition Workshop 0* (2008), 1–8.

[WK03] Wöhler C., Krüger L.: A contour-based stereovision algorithm for video surveillance applications. In *Proceedings of VCIP'03* (2003), pp. 102–109.

[WKE99] Westermann R., Kobbelt L., Ertl T.: Real-time exploration of regular volume data by adaptive reconstruction of iso-surfaces. *The Visual Computer 15* (1999), 100–111.

[WVG92] Wilhelms J., Van Gelder A.: Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3 (July 1992), 201–227.

[WW94] Welch W., Witkin A.: Free-form shape design using triangulated surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), SIGGRAPH '94, ACM, pp. 247–256.

[WWHL04] Wang J., Wong T.-T., Heng P.-A., Leung C.-S.: Discrete wavelet transform on gpu. In *Proceedings of ACM Workshop on General Purpose Computing on Graphic Processors* (2004), pp. C–41.

[ZBK02] Zickler T. E., Belhumeur P. N., Kriegman D. J.: Helmholtz stereopsis: Exploiting reciprocity for surface reconstruction. In *Proceedings of 7th European Conference on Computer Vision* (May 2002), vol. 3, pp. 869–884.

[ZHK*03] Zickler T. E., Ho J., Kriegman D. J., Ponce J., Belhumeur P. N.: Binocular helmholtz stereopsis. In *Proceedings of IEEE Conference on Computer Vision* (October 2003), pp. 1411–1417.

[ZSS97] Zorin D., Schröder P., Sweldens W.: Interactive multiresolution mesh editing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 259–268.

[ZW94] Zabih R., Woodfill J.: Non-parametric local transforms for computing visual correspondence. In *Proceedings of the third European conference on Computer Vision (Vol. II)* (1994), ECCV '94, Springer-Verlag New York, Inc., pp. 151–158.